EXHIBIT C

C1

Fri Jan 04 14:16:53 2008 Function Index Page 2

Fri Jan 04 14:16:53 2008 Function Index Page 1

Fri Jan 04 14:16:53 2008 Function Index Page 2

Fri Jan 04 14:16:53 2008 File Index Page 2

Fri Jan 04 14:16:53 2008 File Index Page 1

Fri Jan 04 14:16:53 2008 File Index Page 2

```
/*
** Copyright 1996,1997 EMC Corporation
*/

/*
** EDMmain.c
**
** Mission Statement: This is the main service file for the EDMsession daemon.
**                    This file contains the main loop, and all calls required
**                    to prepare the daemon to go off and service RPC's.
**
** Primary Data Acted On:
**
** Compile-Time Options:
**
**                      USE_SUNRPC - Compile source with sunrpc support.  If
**                                   not set, assume DCE support.
**
**                      NONPRODUCTION - Compile source for in house, developer
**                                      testing on local work station. Should
**                                      only be used for targted testing.
**
** Basic idea here: Initalize required locks, establish signal handlers,
**                  register RPC interface, go wait for RPCs.
*/

/*
** The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/
#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile$ "
                            "$Revision$ "
                            "$Date$" ;
#endif

/* #define _POSIX_SOURCE   unable to compile with this define set */
/* #define _XOPEN_SOURCE   unable to compile with this define set */

/*****************************************************************************
**
** Routine: main
**
** Inputs: argc, argv
**
** Outputs:  None
**
** Return Codes:
**              exit status
**
** Purpose:     This is the main routine which sets up the daemon
**               to handle RPC calls, and handles them until it is told
**               to stop or it sees a fatal error.
**
** Intended caller:  None
**
```



```
*****************************************************************************
*/
main (int argc, char *argv[])
{
  /*
  ** Parse options
  */

  (void) parse_commandline(argc, argv);

  /*
  ** Setup logging
  */

  (void) daemon_initialize_logging();

  /*
  ** Enable permanent interrupt catching
  */

  (void) daemon_catch_interrupts();

  /*
  ** Function may not return if improper user running daemon
  */

  (void) daemon_check_proper_ID();

  /*
  ** Function will not return if this fails
  */

  (void) daemon_become_daemon();

  /*
  ** Re-establish log initialization since all "fd's" were
  ** closed by esl_daemon_startup (in daemon_become_daemon)
  */

  (void) daemon_initialize_logging();

  /*
  ** This function doesn't return on failure
  */

  (void) daemon_specific_initialization();

  /*
  ** Unregister service, cleanup cache... Never returns...
  */

  (void) daemon_cleanup();

  /*
  **  Strictly to inhibit compiler warning...
  */

  return( 0 );
}
```

```
/*
** Copyright 1996,1997 EMC Corporation
*/

/*
** EDMRestoreEng.c
**
** Mission Statement: This is the main service file for the EDMsessd
                                                        daemon. This
**                    file contains the callbacks from the main
                                                       function which
**                    prepares the daemon to go off and service RPC's.
**
** Primary Data Acted On:
**
** Compile-Time Options:
**
**                      USE_SUNRPC - Compile source with sunrpc
                                                       support.  If
**                                   not set, assume DCE support.
**
** Basic idea here: Module for UNIX specific daemon initialization
*/

/*
** The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/
#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMsessdd.c,v $ "
                            "$Revision: 1.23 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

/* #define _POSIX_SOURCE   unable to compile with this define set */
/* #define _XOPEN_SOURCE   unable to compile with this define set */

#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/inout.h>

#include <stdarg.h>
#include <string.h>
#include <syslog.h>
#include <pthread.h>
#include <thread.h>
#include <sys/utsname.h>
#include <netdb.h>

#include <logging/logging.h>
#include <util/esl_core.h>
#include <util/esl_pidfile.h>
#include <util/esl_daemon.h>
#include <csc/cscomm.h>

#include <restore/csc_EDMRestoreEng.h>

#include <EDMmain.h>
#include <EDMRestoreEngLog.h>
#include <EDMProcessManager.h>
#include <EDMProgress.h>
#include <EDMRE_ccr.h>
#include <EDMRE_ccw.h>
#include <EDMRECommandApi.h>
#include <EDMREQuestionApi.h>
#include <EDMREDrainApi.h>
```



```
/*
 * Need to define _XOPEN_SOURCE for signal funtion definitions
 * and certain signal structure definitions.
 */
#define _XOPEN_SOURCE

#include <signal.h>

#undef _XOPEN_SOURCE

static rpc_if_handle_t if_spec;

static int       G_debug = FALSE;
                                /* Variable which will disable forking */

static char      **commandlineargs; /* Pointer to command line args */

/*****************************************************************************
                                                                    ********
**
** Routine: IsDebugOn
**
** Inputs: None
**
** Outputs:  None
**
** Return Codes:
**              TRUE if debug is on.
**
** Purpose:     This routine can be used to tell other subsystems
**              whether debugging is available.
**
** Intended caller:  internal only.
**
*****************************************************************************
                                                                    ********
*/
boolean_ty
IsDebugOn()
{
#ifdef DEBUG
    return TRUE;
                        /* if DEBUG defined, we must be in debug mode */
#endif
    if (debugmode)        /* if turned on manually via adb, its on */
        return TRUE;

    return G_debug;
                    /* default is how we were started: -d menas debug */
}
```

```
111  /*****************************************************************************
                                                                          ********
112  **
113  ** Routine:kill_handler
114  **
115  ** Inputs: int sigval - the signal which was receieved.
116  **
117  ** Outputs:  Will log messages telling what action is being taken.
118  **
119  ** Return Codes:
120  **               exits with the number of the signal received
121  **
122  ** Purpose:      This routine handles specific signals i.e. SIGINT,
123  **               SIGQUIT,
                           SIGTERM. Each results in a log entry and an exit.
124  **
125  ** Intended caller:  internal only.
126  **
127  *****************************************************************************
                                                                          ********
128  */
129  static void kill_handler( IN int sigval )
130 1 {
131 1 error_status_t   status;
132 1 time_t           current_time;
133 1 char             *ctimebuf;
134 1 char             *ebuff = NULL;

136 1   /* If main exits, it calls this routine with signal 0 */

138 1   /* Unregister the interface */
139 1   (void) csc_unregister_server_interface(&if_spec, &status);

141 1   /* If the unregister fails, report the problem, but continue */
142 1   if ( status != error_status_ok )
143 2   {
144 2     ebuff = (char *) csc_get_error( status );

146 2     (void) EDMRestoreEng_logent(
                       __FILE__,__LINE__, LOG_ERR, MESSAGE_NO_LOGIN, 0,
147 2                    "CSC_SERVER_LOGIN failed: <%d> %s",
148 2                    status, (ebuff ? ebuff : "Unknown error") );
149 1   }

151 1   /* Get the current time */
152 1   (void) time(&current_time);

154 1   ctimebuf = ctime(&current_time);

156 1   /* Overlay newline with null - buf should always be 26 bytes long */
157 1   ctimebuf[ strlen(ctimebuf) - 1 ] = 0;

159 1   (void) EDMRestoreEng_logent(
                     __FILE__,__LINE__, LOG_INFO, MESSAGE_SHUTDOWN, 0,
160 1                "Shutting down at %s due to signal %d", ctimebuf,
                                                              sigval);

162 1   exit(sigval);

164  } /* End of kill_handler() */
```



```
167  /*****************************************************************************
                                                                          ********
168  **
169  ** Routine: unregister_csc
170  **
171  ** Inputs: none
172  **
173  ** Outputs:  Will log messages telling what action is being taken.
174  **
175  ** Return Codes:
176  **               none
177  **
178  ** Purpose:      This routine handles the csc_unregister call
179  **
180  ** Intended caller:  internal and process manager before exit
181  **
182  *****************************************************************************
                                                                          ********
183  */
184  void unregister_csc( void )
185 1 {
186 1 error_status_t   status;
187 1 char     *ebuff = NULL;

189 1   /* Unregister the interface */
190 1   (void) csc_unregister_server_interface(&if_spec, &status);

192 1   /* If the unregister fails, report the problem, but continue */
193 1   if ( status != error_status_ok )
194 2   {
195 2     ebuff = (char *) csc_get_error( status );

197 2     (void) EDMRestoreEng_logent(          __FILE__,__LINE__, LOG_ERR,
198 2                                 MESSAGE_CANNOT_UNREGISTER, 0,
199 2                                 "CSC_UNREGISTER_SERVER failed: <%d> %s",
200 2                                 status, (ebuff ? ebuff : "Unknown error") );
201 1   }

203 1   return;
204  }
```

```
206  /*****************************************************************************
                                                                          ********
207   * Function Name:
208   *       display_usage
209   *
210   * Simply displays the usage
211   *
212   * Call Arguments:
213   *       Program name
214   *
215   * Error Outputs and Side Effects:
216   *       Prints usage.
217   *
218   * Special Considerations:
219   *       None.
220   *
221   *****************************************************************************
                                                                          ********
222   */
223  static void
224  display_usage (IN char *progname)
225 1 {
226 1   /* Print out usage stmt.  */

228 1   fprintf (stderr,"Usage: %s [-d]\n", progname);
229 1   fprintf (
       stderr,"-d keep the daemon from forking so debugging is easier\n");

231  } /* end display_usage () */
```



```
234  /*****************************************************************************
                                                                          ********
235  **
236  ** Routine: daemon_catch_interrupts
237  **
238  ** Inputs:      None
239  **
240  ** Outputs:     None
241  **
242  ** Return Codes:
243  **              None
244  **
245  ** Purpose:     Sets up signals for service. On NT we will have to
246  **              consider what OS constructs to replace signals with.
247  **              In this case we are catching SIGTERM, SIGINT, and
248  **              SIGQUIT and ignoring anything else.
249  **
250  ** Intended caller:  internal only.
251  **
252  *****************************************************************************
                                                                          ********
253  */
254  void daemon_catch_interrupts()
255 1 {
256 1 struct sigaction         sactions;       /* Signal actions */

258 1          ZERO( sactions );

260 1         /*
261 1          * Set an empty list so we can set signals we want to handle
262 1          */
263 1          (void) sigemptyset( &sactions.sa_mask );

265 1         /*
266 1          * Add signals that we want to handle
267 1          */
268 1          (void) sigaddset( &sactions.sa_mask, SIGTERM );
269 1          (void) sigaddset( &sactions.sa_mask, SIGINT );
270 1          (void) sigaddset( &sactions.sa_mask, SIGQUIT );

272 1          /* Setup the signal handler. */
273 1          sactions.sa_handler = kill_handler;

275 1         /*
276 1          * Assign handler to each signal we are interested in.
277 1          */
278 1          (void) sigaction( SIGTERM, &sactions, NULL );
279 1          (void) sigaction( SIGINT,  &sactions, NULL );
280 1          (void) sigaction( SIGQUIT, &sactions, NULL );

282 1          /*
283 1          * Setup mask so we can specify what signals we will ignore.
284 1          */
285 1          (void) sigfillset( &sactions.sa_mask );

287 1         /*
288 1          * We want to ignore everything except those we have set up
289 1          * above so remove those from the list.
290 1          */
291 1          (void) sigdelset( &sactions.sa_mask, SIGTERM );
292 1          (void) sigdelset( &sactions.sa_mask, SIGINT );
293 1          (void) sigdelset( &sactions.sa_mask, SIGQUIT );

295 1         /*
```

```
296 1           * Set the mask. Since no other threads have been started,
297 1           * all threads will get this mask.
298 1           */
299 1           (void) thr_sigsetmask( SIG_SETMASK, &sactions.sa_mask, NULL );
300     }
```



```
303     /*****************************************************************************
                                                                              ********
304     **
305     ** Routine: daemon_check_proper_ID
306     **
307     ** Inputs:      None
308     **
309     ** Outputs:     None
310     **
311     ** Return Codes:
312     **              exits with an error when the user is not root
313     **
314     ** Purpose:     Checks user's ID and determines if the user is allowed
315     **              to execute service.
                                    If there are no constraints then this
316     **              function may be blank.
317     **
318     ** Intended caller:  internal only.
319     **
320     *****************************************************************************
                                                                              ********
321     */

323     void daemon_check_proper_ID()
324 1   {
325 1           /*
326 1           ** Check for root
327 1           */

329 1           if (geteuid() != E_ROOTUID)
330 2           {
331 2             (void) EDMRestoreEng_logent(
                          __FILE__,__LINE__, LOG_ERR, DAEMON_NOTSUPERUSER, 0,
332 2                         "Must be run as superuser, uid was %d",
333 2                         geteuid());
334 2             exit(1);
335 1           }
336     }
```

```
338  /*****************************************************************************
                                                                          ********
339  **
340  ** Routine: parse_commandline
341  **
342  ** Inputs:      argc, argv (command line arguments)
343  **
344  ** Outputs:     None
345  **
346  ** Return Codes:
347  **              exits with an error when the user types a bad argument
348  **
349  ** Purpose:     Parses command line arguments and sets flags. If there
350  **              are no flags to be set then this function may be empty.
351  **
352  ** Intended caller:  internal only.
353  **
354  *****************************************************************************
                                                                          ********
355  */

357  void parse_commandline(int argc, char *argv[])
358 1 {
359 1 int                      opt;            /* Process options   */

361 1          commandlineargs = argv;

363 1          while ((opt = getopt(argc,argv,"dD")) != EOF )
364 2          {
365 2            switch(opt)
366 3            {
367 3              case 'd':
368 3              case 'D':
369 3                  G_debug = TRUE;
370 3                  debugmode = 1;
                                          /* turn on other debugmode flag */
371 3                  break;
372 3              default:
373 3                  (void) display_usage( argv[0] );
374 3                  exit(1);

376 2            }
377 1          }
378  }
```



```
380  /*****************************************************************************
                                                                          ********
381  **
382  ** Routine: daemon_initialize_logging
383  **
384  ** Inputs:      None
385  **
386  ** Outputs:     None
387  **
388  ** Return Codes:
389  **              None
390  **
391  ** Purpose:     Do whatever it takes to initialize logging.
                                                                  In the near
392  **              future this may involve doing something with catalogs
                                                                        or
393  **              calling higher level logging functions which
                                                                  encapsulate
394  **              these things.
395  **
396  ** Intended caller:  internal only.
397  **
398  *****************************************************************************
                                                                          ********
399  */

401  void
402  daemon_initialize_logging()
403 1 {
404 1          /* Pass in argv[0], the program name */
405 1          (void) esl_log_init(commandlineargs[0]);
406  }
```

```
408  /*****************************************************************************
                                                                        ********
409  **
410  ** Routine: daemon_become_daemon
411  **
412  ** Inputs:       None
413  **
414  ** Outputs:      None
415  **
416  ** Return Codes:
417  **               exits with an error code if initialization fails
418  **
419  ** Purpose:      This function is for doing the forking etc. under UNIX.
420  **               It is unknown what will be necessary under NT.
421  **
422  ** Intended caller:  internal only.
423  **
424  *****************************************************************************
                                                                        ********
425  */

427  void
428  daemon_become_daemon()
429 1 {
430 1          char *ptr;
431 1         int  ret = 0;

433 1         /*
434 1          * Strip the path from the program name so we can use it
435 1          * elsewhere.
436 1          */
437 1         ptr = strrchr(commandlineargs[0], '/');
438 1         if (ptr == NULL)
439 1             ptr = commandlineargs[0];
440 1         else
441 1             ptr++;

443 1         /* Change directory to a process specific core directory */
444 1         ret = esl_coredir_setup(ptr);
445 1         if (ret != 0)
446 2         {
447 2              (void) EDMRestoreEng_logent( __FILE__,__LINE__,LOG_ERR,
448 2                                  MESSAGE_ERR_IN_ESL_COREDIR, errno,
449 2                                  "esl_coredir_setup failed" );
450 2             exit(1);
451 1         }

453 1          /*
454 1          **  This is now esl functionality.
                                      This code does everything necessary
455 1          **  to make this a "real" daemon by detaching from the
                                                                    terminal
456 1          **  changing the process group, closing stdout, stderr, stdin,
                                                                        ...
457 1          */

459 1 /***     if (G_debug == FALSE)
460 1         {
461 1             ret = esl_daemon_startup();
462 1              if (ret != 0)
463 1             {
464 1                 fprintf(
                          stderr, "%s: Failed to initialize as daemon.\n",
465 1                       commandlineargs[0]);
466 1                 exit(1);
```



```
467 1             }
468 1         }   **/
469  }
```

```
471  /*****************************************************************************
                                                                          ********
472  **
473  ** Routine: rpc_init
474  **
475  ** Inputs:      None
476  **
477  ** Outputs:     None
478  **
479  ** Return Codes:
480  **              exits with an error code if initialization fails
481  **
482  ** Purpose:     This function is for doing RPC initialization.
483  **              For the most part it involves calling the csc routines.
484  **              This is pretty standard between UNIX and NT.
485  **
486  ** Intended caller:  internal only.
487  **
488  *****************************************************************************
                                                                          ********
489  */

491  void rpc_init()
492 1 {
493 1 error_status_t           status;
                                        /* error status (nbase.h) */
494 1 unsigned char    *conn_h;
495 1 struct hostent  *hp;
496 1 char   *ebuff;
497 1 struct utsname name;

499 1 /*
500 1 ** This is here because of HP which may or may not define timeval.
501 1 ** May be removed when esl_timeval is ported to clients
502 1 */
503 1 #ifdef _STRUCT_TIMEVAL
504 1         struct timeval  sleep_interval = {5,0};
                                        /* 5 second sleep interval */
505 1 #else
506 1         struct timespec sleep_interval = {5,0};
                                        /* 5 second sleep interval */
507 1 #endif

509 1         /* Setup the interface specification for RPC */

511 1         RE_SERVER_IFSPEC(if_spec);

513 1         /*
514 1         * Login as SERVER_PRINCIPAL.  The context of the process
515 1         * will be set to this principal.
516 1         *
517 1         * This process will keep trying to login to DCE if the
                                                                  registery
518 1         * server is unavailable.
                                Note that under SUN RPC this is a no-op.
519 1         */
520 1         while (TRUE)
521 2         {
522 2                 (void) csc_server_login(RE_SERVER_PRINCIPAL,
523 2                                         RE_SERVER_KEYTAB, &status);

525 2                 /* If we succeeded, then exit this loop.  */
526 2                 if ( status == error_status_ok )
527 3                 {
528 3                         break;
```



```
529 2                 }
530 2                 else    /* Print error message if appropriate. */
531 3                 {
532 3                  ebuff = (char *) csc_get_error( status );

534 3                    (void) EDMRestoreEng_logent(
                                                __FILE__,__LINE__,LOG_ERR,
535 3                                       MESSAGE_NO_LOGIN, 0,
536 3                                       "CSC_SERVER_LOGIN failed: <%d>
                                                                      %s",
537 3                                       status, (
                                          ebuff ? ebuff : "Unknown error"));
538 2                 }

540 2                 /* If the failure was due to unavailable client,
541 2                 * pause and then try again.
542 2                 */
543 2                 if (status == sec_rgy_server_unavailable)
544 3                 {
545 3                         /*
546 3                          * uses sleep when SUNRPC, otherwise uses
547 3                          * pthread call to delay for the specified
548 3                          * time
549 3                          */
550 3                            CSC_SLEEP(sleep_interval);
551 3                            continue;
552 2                 }

554 2                 /* If we got here, we had a unexpected failure.  */
555 2                 (void) EDMRestoreEng_logent(
                                                __FILE__,__LINE__,LOG_ERR,
556 2                                       MESSAGE_NO_LOGIN, 0,
557 2                                      "The service cannot log in as
                                                                  required");


559 2                 exit(1);
560 1          }

562 1         uname(&name);

564 1         hp = gethostbyname(name.nodename);

566 1         if (hp == NULL)
567 2         {
568 2              (void) EDMRestoreEng_logent( __FILE__,__LINE__,LOG_ERR,
569 2                                           MESSAGE_GETHOSTBYNAME_FAIL,
                                                                  errno,
570 2                                           "gethostbyname failed" );
571 2              exit(1);
572 1         }

574 1         memcpy((char *) &if_spec.ip_addr, hp -> h_addr, hp->h_length);

576 1          /*
577 1          ** We need to initialize the authorization module before we
                                                                  do
578 1          ** a listen.
579 1          */
580 1          (void)csc_authorization_init(&status);

582 1          if ( status != error_status_ok )
583 2          {
584 2           ebuff = (char *) csc_get_error( status );

586 2             (void) EDMRestoreEng_logent( __FILE__,__LINE__,LOG_ERR,
587 2                                          MESSAGE_NOAUTHORIZATION, 0,
```

```
588 2                                "CSC_AUTHORIZATION_INIT failed: <%d> %s",
589 2                                 status, (
                                           ebuff ? ebuff : "Unknown error") );
590 2              exit(1);
591 1            }

593 1           conn_h = calloc(1, CONNECT_HANDLE_SIZE);

595 1           if (conn_h == NULL)
596 2           {
597 2              (void) EDMRestoreEng_logent( __FILE__,__LINE__,LOG_ERR,
598 2                                  MESSAGE_NO_MEMORY, 0,
599 2                                  "Failure allocating memory for connection
                                                                      handle");
600 2              exit(1);
601 1           }

603 1            (void) csc_register_private_server_interface(
                                                    &if_spec,
604 1                                               0,
605 1                                               1,
606 1                                               conn_h,
607 1                                               &status);

609 1            if ( status != error_status_ok )
610 2            {
611 2             ebuff = (char *) csc_get_error( status );

613 2              (void) EDMRestoreEng_logent( __FILE__,__LINE__,LOG_ERR,
614 2                                  MESSAGE_CANNOTREGISTER, 0,
615 2                                 "CSC_REGISTER_SERVER_INTERFACE failed:
                                                                      <%d> %s",
616 2                                  status, (
                                           ebuff ? ebuff : "Unknown error") );
617 2              exit(1);
618 1            }

620 1            free(conn_h);
621    }
```



```
623    /*************************************************************************
                                                                        ********
624    **
625    ** Routine: rpc_run
626    **
627    ** Inputs:       None
628    **
629    ** Outputs:      None
630    **
631    ** Return Codes:
632    **               None
633    **
634    ** Purpose:      This function is for running the RPC listen.
635    **               This is pretty standard between UNIX and NT.
636    **
637    ** Intended caller:  internal only.
638    **
639    *************************************************************************
                                                                        ********
640    */

642    void rpc_run()
643 1  {
644 1  error_status_t            status;
                                          /* error status (nbase.h) */
645 1  char  *ebuff;

647 1            /* listen for RPC calls forever. */
648 1            (void) csc_server_listen(
                                rpc_c_listen_max_calls_default, &status );

650 1           ebuff = (char *) csc_get_error( status );

652 1            /* We don't expect to get here. */
653 1            (void) EDMRestoreEng_logent( __FILE__,__LINE__,LOG_ERR,
654 1                                MESSAGE_SERVERLISTEN, 0,
655 1                                "CSC_SERVER_LISTEN failed: <%d> %s",
656 1                                 status, (
                                          ebuff ? ebuff : "Unknown error") );
657    }
```

```
659  /*****************************************************************
                                                                ********
660  **
661  ** Routine: daemon_specific_initialization
662  **
663  ** Inputs:      None
664  **
665  ** Outputs:     None
666  **
667  ** Return Codes:
668  **              None
669  **
670  ** Purpose:     Do whatever makes this daemon special.
                                                        In some cases you
671  **              may want to start a thread or open a socket.
                                                          Do that here.
672  **
673  ** Intended caller:  internal only.
674  **
675  *****************************************************************
                                                                ********
676  */

678  void
679  daemon_specific_initialization()
680 1 {
681 1 int                     status;
                                        /* error status (nbase.h) */
682 1 void                    *statptr;
                                        /* error status (nbase.h) */
683 1 int                     ret;
684 1 pthread_t               pmtid;
685 1 pthread_t               progresstid;
686 1 pthread_t               ccrtid;
687 1 pthread_t               ccwtid;
688 1 time_t    current_time;
689 1 char      *ctimebuf;

691 1           ret = CommandAPIInit(&status);
692 1           ret = QuestionAPIInit(&status);
693 1           ret = DrainAPIInit(&status);

695 1           /* Find out what time it is */
696 1           (void) time(&current_time);

698 1           ctimebuf = ctime(&current_time);

700 1           /* Overlay newline with null - buf should always be 26 bytes
                                                                long */
701 1           ctimebuf[ strlen(ctimebuf) - 1 ] = 0;

703 1           /* Log startup message */
704 1           (void) EDMRestoreEng_logent( __FILE__, __LINE__, LOG_INFO,
705 1                           MESSAGE_STARTUP, 0,
                                   "Restore service %s starting up at %s",
706 1                           commandlineargs[0], ctimebuf );

708 1         /*
709 1          * Start the other threads in the daemon. The main thread
710 1          * becomes the RPC thread. REProcessManager is the
711 1          * entry point for the periodic event thread.
712 1          */
713 1         pthread_create(&pmtid, NULL, REProcessManager, NULL);
714 1         pthread_create(&progresstid, NULL, REProgress, NULL);
715 1         /* pthread_create(&ccrtid, NULL, RestoreSvc_ccr, NULL); */
```



```
716 1           pthread_create(&ccwtid, NULL, RestoreSvc_ccw, NULL);


719 1           rpc_init();
720 1           RestoreSvc_Setup();
721 1           rpc_run();

723 1           pthread_join(pmtid, &statptr);
724     }
```

```
726  /*****************************************************************
                                                                ********
727  **
728  ** Routine: daemon_cleanup
729  **
730  ** Inputs:       None
731  **
732  ** Outputs:      None
733  **
734  ** Return Codes:
735  **               None
736  **
737  ** Purpose:      Call function which will clean up daemon properly.
738  **
739  ** Intended caller:  internal only.
740  **
741  *****************************************************************
                                                                ********
742  */

744  void
745  daemon_cleanup()
746 1 {
747 1          kill_handler( 0 );
748  }
```

```
/*
** Copyright 1996,1997 EMC Corporation
*/

/*
** EDMProcessManager.c
**
** Mission Statement: This is the entry point for the Process Manager
                                                                thread.
**
** Primary Data Acted On:
**
** Compile-Time Options:
**
**                      USE_SUNRPC - Compile source with sunrpc
                                                          support.  If
**                                    not set, assume DCE support.
**
** Basic idea here: Module for coding the Process Manager thread.
*/

/*
** The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/
#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMProcessManager.c,v $ "
                            "$Revision: 1.23 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

/* #define _POSIX_SOURCE   unable to compile with this define set */
/* #define _XOPEN_SOURCE   unable to compile with this define set */

#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/inout.h>

#include <syslog.h>
#include <unistd.h>
#include <stdlib.h>

#include <pthread.h>
#include <EDMProcessManager.h>
#include <EDMRECommandApi.h>
#include <EDMRestoreEngLog.h>
#include <EDMmain.h>
#include <restore/restore_engine.h>
#include <restore/restore_api.h>
#include <restore/REprogmsg.h>
#include <restore/EDMREProgressApi.h>
#include <EDMFinalStatus.h>


/* local prototypes */

static void unregister_rpc( void );
static void start_completion( EDMREGlobalStatus );


/* local data */

static boolean_ty        completion_signalled = FALSE;

struct timeout_array
{
```



```
    time_t        timeoutlen;
    time_t        guideadlen;
} tval[MAX_GLOBAL_STATUS_VALUE+1] = {
     // Timeout value         Gui is dead value
   { 5*SECONDS_PER_MINUTE, 5*SECONDS_PER_MINUTE }, // Exiting
   { 5*SECONDS_PER_MINUTE, 5*SECONDS_PER_MINUTE }, // Starting
   { SECONDS_PER_YEAR,     2*SECONDS_PER_HOUR },   // Browsing
   { 3*SECONDS_PER_HOUR,   5*SECONDS_PER_MINUTE }, // Pre phase
   { 10*SECONDS_PER_DAY,   5*SECONDS_PER_MINUTE }, // Execute
   { 2*SECONDS_PER_DAY,    5*SECONDS_PER_MINUTE }  // Post Phase
   };

boolean_ty
IsRestoreTimedOut(IN time_t lastguitime, IN time_t incurrentstate,
                  IN int status)
{
    time_t t = time(NULL);

    if (status > MAX_GLOBAL_STATUS_VALUE)
    {
        return FALSE;
    }

    if (status < 0)
    {
        status = 0;      /* all exiting conditions use same timeout */
    }

    if ( (t - tval[status].guideadlen) > lastguitime)
    {
        return TRUE;
    }
    else if ( (t - tval[status].timeoutlen) > incurrentstate)
    {
        return TRUE;
    }

    return FALSE;
}
```

```
105     void *
106     REProcessManager(void *buff)
107 1   {
108 1       int  status;
109 1       int  command;
110 1       int  result;
111 1       void         *input_ptr;
112 1       void         *output_ptr;
113 1       boolean_ty   finish_rpc_recvd = FALSE;
114 1       boolean_ty   reader_finish_rcvd = FALSE;
115 1       EDMREGlobalStatus    internal_status;
116 1       time_t       status_time;

118 1       setGlobalStatus( EDMRE_STATE_STARTING );

120 1       while (
        !reader_finish_rcvd || !finish_rpc_recvd )   /* until time to exit */
121 2       {
122 2           /* wait for next command */
123 2           if (PopCommand( 1, &command, &status ))
124 3           {
125 3               if (COMMAND_RECORD_GET_FAILED != status)
126 3               /* log error if not 'normal' queue empty error */
127 3                   EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
128 3                                         MESSAGE_RECORD_GET_FAILED, 0,
129 3                                         "PopCommand failed;
                                                              status = %d",
130 3                                         status );

132 3               /* check for completion timeout or idle timeout */
133 3               internal_status = getGlobalStatus( &status_time );
134 3               if (completion_signalled)
135 4               {
136 4                   if (TRUE == IsRestoreTimedOut(  getLastRpcTime(
                                                          ), status_time,
137 4                                                 internal_status ) )
138 5                   {
139 5                       if (!finish_rpc_recvd)
140 6                       {
                         /* let restore service module clean up, stop rpcs */
141 6                           result = EDMRE_Finish( NULL, NULL );
142 6                           unregister_rpc( );      /* cleanup csc i/f */
143 5                       }
144 5                       EDMRestoreEng_logent(
                                          __FILE__, __LINE__, LOG_ERR,
145 5                                 MESSAGE_SHUTDOWN, 0,
146 5                                 "Shutting down after timeout awaiting
                                                              sync" );
147 5                       break;
                                                 /* escape while to exit */
148 4                   }
149 3               } else
150 4               {
151 4                   if (TRUE == IsRestoreTimedOut( getLastRpcTime(
                                                          ), status_time,
152 4                                                 internal_status ) )
153 5                   {   /* if already exiting, leave state alone */
154 5                       if (internal_status > 0)
155 5                           start_completion( EDMRE_STATE_TIMEOUT );
156 5                       else
157 5                           start_completion( internal_status );
158 4                   }
159 3               }
160 3               continue;
                      /* keep waiting in case thread wait got interrupted */
```



```
161 2           }

163 2           /* got some command; see if we're in completion sequence: */
164 2           if (completion_signalled)
165 3           {
166 3               if (COMMAND_FINISH == command && !finish_rpc_recvd)
167 4               {
168 4                   if (PopRpcInput( &input_ptr, &status ))
169 5                   {
170 5                       EDMRestoreEng_logent(
                                                  __FILE__, __LINE__, LOG_ERR,
171 5                                      MESSAGE_POP_RPC_INPUT_FAILED, 0,
172 5                                      "PopRpcInput failed;
                                                        status = %d", status );
173 4                   }
174 4                   else
175 4                       /* let restore service module clean up;
                                                              stop rpc's */
176 4                       result = EDMRE_Finish( input_ptr, &output_ptr );

178 4                   finish_rpc_recvd = TRUE;
179 4                   unregister_rpc( );
180 3               }
181 3               else if (
                      COMMAND_READER_FINISHED == command && !reader_finish_rcvd)
182 4               {
183 4                   reader_finish_rcvd = TRUE;
184 5                   if (!finish_rpc_recvd) {
185 5                   /* let restore service module clean up, stop rpcs */
186 5                       result = EDMRE_Finish( NULL, NULL );
187 5                       unregister_rpc( );  /* cleanup csc i/f */
188 4                   }
189 4                   break;          /* exit */
190 3               }
191 3               else
192 4               {
193 4                   EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
194 4                                         MESSAGE_INVALID_COMMAND, 0,
                                                              "cmd value: %d",
195 4                                         command );
196 3               }
197 3               continue;
                      /* check if both finishes recvd, else keep waiting */
198 2           }

200 2           /* not in completion seq;
                                get pointer to rpc input argument structure */
201 2           if (PopRpcInput( &input_ptr, &status ))
202 3           {
203 3               EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
204 3                                     MESSAGE_POP_RPC_INPUT_FAILED, 0,
205 3                                  "PopRpcInput failed;
                                                    status = %d Lost command: %d",
206 3                                     status, command );
207 3               continue;   /* ??? keep trying or return ?? */
208 2           }

210 2           switch( command )
211 3           {
212 3           case COMMAND_GET_RESTORABLE_OBJECTS:
213 3               result = EDMRE_GetRestorableObjects(
                                                      input_ptr, &output_ptr );
214 3               break;
215 3           case COMMAND_MARK_OBJECT:
216 3               result = EDMRE_MarkObject( input_ptr, &output_ptr );
217 3               break;
```

```
218 3           case COMMAND_UNMARK_OBJECT:
219 3               result = EDMRE_UnmarkObject( input_ptr, &output_ptr );
220 3               break;
221 3           case COMMAND_SUBMIT:
222 3               result = EDMRE_Submit( input_ptr, &output_ptr );
223 3               break;

225 3           case COMMAND_START:
226 3               result = EDMRE_Start( input_ptr, &output_ptr );
227 3   #if 0       /* taken out to allow continuation after successful & aborted
                                                                    restore */
228 3               start_completion( getGlobalStatus(NULL) );
                                                    /* leave same state */
229 3   #endif
230 3               break;

232 3           case COMMAND_FIND_RESTORABLE_OBJECTS:
233 3               result = EDMRE_FindRestorableObjects(
                                                input_ptr, &output_ptr );
234 3               break;

236 3           case COMMAND_FINISH:
237 3               result = EDMRE_Finish( input_ptr, &output_ptr );
238 3               finish_rpc_recvd = TRUE;
239 3               if (EDMRE_STATE_SUCCESSFUL
240 3                   < (internal_status = getGlobalStatus(
                                                        &status_time ) )
241 3               )
242 3                   /* if already exiting, leave state alone */
243 3                   start_completion( EDMRE_STATE_SUCCESSFUL );
244 3               else
245 3                   start_completion( internal_status );
246 3               unregister_rpc( );
247 3               break;              /* await dispatcher finish command */

249 3           case COMMAND_LOAD_RECX_DIRECTIVES:
250 3               result = EDMRE_Load_recx_directives(
                                                input_ptr, &output_ptr );
251 3               break;
252 3           case COMMAND_GET_ALL_TIMES:
253 3               result = EDMRE_GetAllBackupTimes(
                                                input_ptr, &output_ptr );
254 3               break;
255 3           case COMMAND_SET_PREVIOUS_BACKUP:
256 3               result = EDMRE_SetPreviousBackup(
                                                input_ptr, &output_ptr );
257 3               break;
258 3           case COMMAND_SET_NEXT_BACKUP:
259 3               result = EDMRE_SetNextBackup( input_ptr, &output_ptr );
260 3               break;
261 3           case COMMAND_SET_FIRST_BACKUP:
262 3               result = EDMRE_SetFirstBackup( input_ptr, &output_ptr );
263 3               break;
264 3           case COMMAND_SET_MOST_RECENT_BACKUP:
265 3               result = EDMRE_SetMostRecentBackup(
                                                input_ptr, &output_ptr );
266 3               break;
267 3           case COMMAND_SET_BACKUP_FOR_TIME:
268 3               result = EDMRE_SetBackupForTime( input_ptr, &output_ptr );
269 3               break;
270 3           default:
271 3               EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
272 3                                     MESSAGE_INVALID_COMMAND, 0,
                                                        "cmd value: %d",
273 3                                     command );
274 3               result = COMMAND_RESULT_FAILURE;
```



```
275 2           }

277 2           /* push result arg structure pointer, IF command succeeded */
278 2           if( result != COMMAND_RESULT_FAILURE)
279 3           {
280 3               if (PushRpcOutput( output_ptr, &status) )
281 4               {
282 4                   EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
283 4                                         MESSAGE_PUSH_RPC_OUTPUT_FAILED,
                                                                    0,
284 4                                         "PushRpcOutput failed;
                                                        status = %d",
285 4                                         status );
286 3               }
287 2           }

289 2           if (PushResult( result, command, &status) )
290 3           {
291 3               EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
292 3                                     MESSAGE_FAILURE_TO_QUEUE_RESULT, 0,

293 3                                     "PushResult failed;
                                                status = %d", status );
294 2           }
295 1       }

297 1   #if 0    /* I think we just leave global status as its already set */
298 1       if ( reader_finish_rcvd && finish_rpc_recvd )        /*  good exit
                                                                    */
299 1           setGlobalStatus( /* good exit ?? */ 0 );
300 1   #endif

302 1       exit ( getGlobalStatus(NULL) );
303 1       return buff;
304     }
```

```
307     /* local function to unregister rpc interface */

309     static void unregister_rpc( void )
310   1 {
311   1         sleep( 1 );
                          /* allow last rpc (finish) response to get sent */
312   1         unregister_csc( );      /* stop RPC traffic */
313   1         return;
314     }
```



```
317     /* local function to start completion sequence */

319     static void start_completion( EDMREGlobalStatus status )
320   1 {
321   1         setGlobalStatus( status );
322   1         SendFinalStatus( );             /* signal dispatcher */
323   1         completion_signalled = TRUE;
324   1         return;
325     }
```

```
/*****************************************************************************
**
** File Name:   EDMREProcMgrService.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
**       This module contains the Process Manager (thread) functions that
**       provide the top level processing of the 'asynchronous' Restore
**       Engine RPC's.  These functions are basically 'wrappers' for the
*        Restore Service Library calls that perform the actual RPC services.
**
** Table of Contents:
** ------------------
**               EDMRE_GetRestorableObjects
**               EDMRE_MarkObject
**               EDMRE_UnmarkObject
**               EDMRE_Submit
**               EDMRE_Start
**               EDMRE_Finish
**               EDMRE_FindRestorableObjects
**
**       Internal Functions:
**               EDMRE_ProgressCallback
**               EDMRE_RestoreCallback
**
** Compile-Time Options:
**
**
*****************************************************************************
*****/



/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif



/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

/* #define  _POSIX_SOURCE 1 */



/*
 * System headers.
 */
#include <stdlib.h>
#include <sys/syslog.h>
```



```
#include <unistd.h>
#include <string.h>

/*
 * Epoch headers.
 */

#include <ebutil/ebutil.h>
#include <restore/restore_engine.h>
#include <restore/REprogmsg.h>
#include <restore/EDMREProgressApi.h>

/*
 * Local headers
 */
#include <RSLapi.h>
#include <EDMProcessManager.h>
#include <EDMRestoreEngLog.h>
#include <EDMRECommandApi.h>



/*
 * Local functions
 */
static boolean_ty EDMRE_ProgressCallback( unsigned long progress );
static boolean_ty EDMRE_RestoreCallback( void );

/*****************************************************************************
**
** Routine:  EDMRE_GetAllBackupTimes
**
** Inputs:   void *input_args    ptr to struct with RPC input args
**
** Outputs:  void **status       addr of void * to receive ptr output arg struct
**
** Return Codes:
**           0 for success and non-zero for failure.
**
** Purpose:  Wrapper of Restore service library call to be executed
**            asynchronously from main RPC thread
**
*****************************************************************************
*/
int EDMRE_GetAllBackupTimes( void *input_args, void **output_args )
{
    RE_get_all_backup_times_args         *in_args
        = (RE_get_all_backup_times_args *)input_args;
    RE_get_all_backup_times_result       *out_args;

    int          status = COMMAND_RESULT_SUCCESS;

    out_args = calloc( 1, sizeof (RE_get_all_backup_times_result) );


    if (NULL == out_args)
    {
        EDMRestoreEng_logent(
                    __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
                0, "calloc fail for RE_get_all_backup_times_args" );
        status = COMMAND_RESULT_FAILURE;        /* fatal error */
    }

    else
```

```
123 2     {
124 2          out_args->cookie = in_args->cookie;
125 2          out_args->status = RSTSL_GetAllBackupTimes(
                                                    in_args->startTime,
                                                      in_args->endTime,
126 2
127 2
                                                    in_args->maxEntries,
                                                       in_args->flags,
128 2
129 2
                                                &out_args->backupTimes,
130 2
                                                 &out_args->numEntries,
131 2
                                                    &out_args->cookie);
132 2          *output_args = (void *) out_args;
133 1     }

135 1     xdr_free( xdr_RE_get_all_backup_times_args, (char *)in_args);
136 1     free( in_args);

138 1     return status;
139     }
```



```
141   /*************************************************************************
                                                                        ********

142   **
143   ** Routine:  EDMRE_GetRestorableObjects
144   **
145   ** Inputs:   void *input_args    ptr to struct with RPC input args
146   **
147   ** Outputs:  void **status       addr of void * to receive ptr output
                                                              arg struct
148   **
149   ** Return Codes:
150   **           0 for success and non-zero for failure.
151   **
152   ** Purpose:  Wrapper of Restore service library call to be executed
153   **           asynchronously from main RPC thread
154   **
155   *************************************************************************
                                                                        ********

156   */
157   int EDMRE_GetRestorableObjects( void *input_args, void **output_args )
158 1 {
159 1     RE_get_restorable_objects_start_args         *in_args
160 1          = (RE_get_restorable_objects_start_args *)input_args;
161 1     RE_get_restorable_objects_output_result      *out_args;

163 1     int          status = COMMAND_RESULT_SUCCESS;

165 1     out_args = calloc( 1, sizeof(
                                RE_get_restorable_objects_output_result) );
166 1     if (NULL == out_args)
167 2     {
168 2          EDMRestoreEng_logent(
                              __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
169 2                    0,
                "calloc fail for RE_get_restorable_objects_output_result" );
170 2          status = COMMAND_RESULT_FAILURE;        /* fatal error */
171 1     }

173 1     else
174 2     {
175 2          out_args->cookie = in_args->cookie;
176 2          out_args->status = RSTSL_GetRestorableObjects(
177 2          (restorableObjectPtr)in_args->parentObj->RE_restorable_obj_u.
                                                                    tloInfo,
178 2                                          in_args->parentObj->objLevel,
179 2                                          &out_args->childrenObjs,
180 2                                          &out_args->cookie,
181 2                                          in_args->maxEntries,
182 2                                          &out_args->numEntries,
183 2                                          in_args->allowBadFiles );
184 2          *output_args = (void *)out_args;
185 1     }

187 1     xdr_free( xdr_RE_get_restorable_objects_start_args, (
                                                          char *)in_args);
188 1     free( in_args);

190 1     return status;
191   }
```

```
193   /*****************************************************************************
                                                                          ********
194   **
195   ** Routine:  EDMRE_MarkObject
196   **
197   ** Inputs:   void *input_args    ptr to struct with RPC input args
198   **
199   ** Outputs:  void **status       addr of void * to receive ptr output
                                                                    arg struct
200   **
201   ** Return Codes:
202   **           0 for success and non-zero for failure.
203   **
204   ** Purpose:  Wrapper of Restore service library call to be executed
205   **            asynchronously from main RPC thread
206   **
207   *****************************************************************************
                                                                          ********
208   */
209   int EDMRE_MarkObject( void *input_args, void **output_args )
210 1 {
211 1     RE_mark_object_args          *in_args = (
                                           RE_mark_object_args *)input_args;
212 1     RE_get_mark_results_result   *out_args;

214 1     int           status = COMMAND_RESULT_SUCCESS;

216 1     out_args = calloc( 1, sizeof(RE_get_mark_results_result) );
217 1     if (NULL == out_args)
218 2     {
219 2          EDMRestoreEng_logent(
                          __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
220 2                  0, "calloc fail for RE_get_mark_results_result" );
221 2          status = COMMAND_RESULT_FAILURE;        /* fatal error */
222 1     }

224 1     else
225 2     {
226 2          out_args->status = RSTSL_MarkObject( in_args->thisObj,
227 2                                               in_args->backupTime,
228 2                                               in_args->allowBadFiles,
229 2                                               in_args->descend,
230 2                                               &out_args->badFileCount,
231 2
                                            &out_args->permDenyFileCount,
232 2                                               &out_args->fileMarkCount,
233 2                                               &out_args->dirMarkCount,
234 2
                                              &out_args->otherMarkCount,
235 2                                             EDMRE_ProgressCallback );
236 2          *output_args = (void *)out_args;
237 1     }

239 1     xdr_free( xdr_RE_mark_object_args, (char *)in_args);
240 1     free( in_args );

242 1     return status;
243   }
```



```
245   /*****************************************************************************
                                                                          ********
246   **
247   ** Routine:  EDMRE_UnmarkObject
248   **
249   ** Inputs:   void *input_args    ptr to struct with RPC input args
250   **
251   ** Outputs:  void **status       addr of void * to receive ptr output
                                                                    arg struct
252   **
253   ** Return Codes:
254   **           0 for success and non-zero for failure.
255   **
256   ** Purpose:  Wrapper of Restore service library call to be executed
257   **            asynchronously from main RPC thread
258   **
259   *****************************************************************************
                                                                          ********
260   */
261   int EDMRE_UnmarkObject( void *input_args, void **output_args )
262 1 {
263 1     RE_unmark_object_args          *in_args = (
                                           RE_unmark_object_args *)input_args;
264 1     RE_get_unmark_results_result *out_args;

266 1     int           status = COMMAND_RESULT_SUCCESS;

268 1     out_args = calloc( 1, sizeof(RE_get_unmark_results_result) );
269 1     if (NULL == out_args)
270 2     {
271 2          EDMRestoreEng_logent(
                          __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
272 2                  0, "calloc fail for RE_get_unmark_results_result" );
273 2          status = COMMAND_RESULT_FAILURE;        /* fatal error */
274 1     }

276 1     else
277 2     {
278 2          out_args->status = RSTSL_UnmarkObject( in_args->thisObj,
279 2                                                 in_args->backupTime,
280 2                                                 in_args->badFilesOnly,
281 2                                                 in_args->descend,
282 2                                                 &out_args->badFileCount,
283 2                                                 &out_args->fileMarkCount,
284 2                                                 &out_args->dirMarkCount,
285 2
                                                &out_args->otherMarkCount,
286 2                                                 EDMRE_ProgressCallback );
287 2          *output_args = (void *)out_args;
288 1     }

290 1     xdr_free( xdr_RE_unmark_object_args, (char *)in_args);
291 1     free( in_args );

293 1     return status;
294   }
```

```
296  /*************************************************************************
                                                                        ********
297  **
298  ** Routine: EDMRE_ProgressCallback
299  **
300  ** Inputs:   unsigned long progress      objects processed so far
301  **
302  ** Outputs:  none
303  **
304  ** Return Codes:
305  **           boolean_ty          FALSE if operation can continue
306  **                               TRUE if operation should be cancelled
307  **
308  ** Purpose:  Restore service library callback function to be called
                                                                  periodically
309  **            to return progress information and check for
                                                                 cancellation.
310  **
311  *************************************************************************
                                                                        ********
312  */
313  static boolean_ty EDMRE_ProgressCallback( unsigned long progress )
314 1 {
315 1          UpdateProgressValue( progress );
316 1          return TestRpcCancelFlag( );
317  }
```



```
319  /*************************************************************************
                                                                        ********
320  **
321  ** Routine: EDMRE_RestoreCallback
322  **
323  ** Inputs:   none
324  **
325  ** Outputs:  none
326  **
327  ** Return Codes:
328  **           boolean_ty          FALSE if operation can continue
329  **                               TRUE if operation should be cancelled
330  **
331  ** Purpose:  Restore service library callback function to be called
                                                                  periodically
332  **            by 'Start' function to return check for cancellation.
333  **
334  *************************************************************************
                                                                        ********
335  */
336  static boolean_ty EDMRE_RestoreCallback( void )
337 1 {
338 1          EDMREGlobalStatus       internal_status;
339 1          long    last_rpc_time;
340 1          time_t  status_time;

342 1          internal_status = getGlobalStatus( &status_time );
343 1          last_rpc_time = getLastRpcTime( );

345 1          if (internal_status == EDMRE_STATE_USER_QUIT
346 1           || internal_status == EDMRE_STATE_ADMIN_QUIT)  /* someone
                                                                  aborted */
347 1                  return TRUE;
348 1          if (TRUE == IsRestoreTimedOut( last_rpc_time, status_time,
349 1                                         internal_status )
350 1          )
351 1                  return TRUE;
                                                    /* no sign of user life */
352 1          return TestRpcCancelFlag( );
                                                   /* user-signalled cancel */
353  }
```

```
355  /*****************************************************************************
                                                                          ********
356  **
357  ** Routine:  EDMRE_Submit
358  **
359  ** Inputs:   void *input_args    ptr to struct with RPC input args
360  **
361  ** Outputs:  void **status       addr of void * to receive ptr output
                                                                  arg struct
362  **
363  ** Return Codes:
364  **           0 for success and non-zero for failure.
365  **
366  ** Purpose:  Wrapper of Restore service library call to be executed
367  **            asynchronously from main RPC thread
368  **
369  *****************************************************************************
                                                                          ********
370  */
371  int EDMRE_Submit( void *input_args, void **output_args )
372 1 {
373 1     RE_submit_args               *in_args = (
                                                RE_submit_args *)input_args;
374 1     RE_get_submit_results_output         *out_args;
375 1     unsigned int                 object_count = 0;
376 1     EDMRST_submit_args           *submitArgs = calloc(1,sizeof(
                                                    EDMRST_submit_args));
377 1     int          status = COMMAND_RESULT_SUCCESS;

379 1     out_args = calloc( 1, sizeof(RE_get_submit_results_output) );
380 1     if (NULL == out_args)
381 2     {
382 2          EDMRestoreEng_logent(
                          __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
383 2                  0, "calloc fail for RE_get_submit_results_output" );
384 2          status = COMMAND_RESULT_FAILURE;        /* fatal error */
385 1     }

387 1     else
388 2     {
389 2         submitArgs->clientSocketPort = in_args->socketPort;
390 2          submitArgs->mapfile_env = esl_strdup(in_args->mapFile_env);
391 2          submitArgs->socketClientNm = esl_strdup(
                                             in_args->socketClientName);

393 2          out_args->submitObjectID = in_args->submitObjectID;
394 2          out_args->status = RSTSL_Submit( in_args->hostname,
395 2                                           in_args->overwritePolicy,
396 2                                           in_args->inPlace,
397 2                                           in_args->directory,
398 2                                           in_args->transport,
399 2                                           &out_args->submitObjectID,
400 2                                           &object_count,
401 2                                           EDMRE_ProgressCallback,
402 2                                           submitArgs);
403 2          out_args->objectsDone = object_count;
404 2          *output_args = (void *)out_args;
405 1     }

407 1     xdr_free( xdr_RE_submit_args, (char *)in_args);
408 1     free( in_args );

410 1     return status;
411   }
```



```
413  /*****************************************************************************
                                                                          ********
414  **
415  ** Routine:  EDMRE_Start
416  **
417  ** Inputs:   void *input_args    ptr to struct with RPC input args
418  **
419  ** Outputs:  void **status       addr of void * to receive ptr output
                                                                  arg struct
420  **
421  ** Return Codes:
422  **           0 for success and non-zero for failure.
423  **
424  ** Purpose:  Wrapper of Restore service library call to be executed
425  **            asynchronously from main RPC thread
426  **
427  *****************************************************************************
                                                                          ********
428  */
429  int EDMRE_Start( void *input_args, void **output_args )
430 1 {
431 1     RE_start_args        *in_args = (RE_start_args *)input_args;
432 1     RE_status_result     *out_args;

434 1     int          status = COMMAND_RESULT_SUCCESS;

436 1     out_args = calloc( 1, sizeof(RE_status_result) );
437 1     if (NULL == out_args)
438 2     {
439 2          EDMRestoreEng_logent(
                          __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
440 2                  0, "calloc fail for RE_status_result" );
441 2          status = COMMAND_RESULT_FAILURE;        /* fatal error */
442 1     }

444 1     else
445 2     {
446 2          out_args->status = RSTSL_Start( in_args->submitObjectID,
447 2                                          EDMRE_RestoreCallback );
448 2          *output_args = (void *)out_args;
449 1     }

451 1     xdr_free( xdr_RE_start_args, (char *)in_args);
452 1     free( in_args );

454 1     return status;
455   }
```

```
457  /*****************************************************************************
                                                                          ********
458  **
459  ** Routine:  EDMRE_Finish
460  **
461  ** Inputs:   void *input_args     OPTIONAL ptr to struct with RPC input
                                                                             args
462  **
463  ** Outputs:  void **status       OPTIONAL addr of void * to receive
                                                                           ptr to
464  **                               output arg struct
465  **
466  ** Return Codes:
467  **           0 for success and non-zero for failure.
468  **
469  ** Purpose:  Wrapper of Restore service library call to be executed
470  **            asynchronously from main RPC thread
471  **
472  *****************************************************************************
                                                                          ********
473  */
474  int EDMRE_Finish( void *input_args, void **output_args )
475 1 {
476 1          int     status = COMMAND_RESULT_SUCCESS;
477 1          RE_status_result        *out_args;

479 1          out_args = calloc( 1, sizeof(RE_status_result) );

481 1          if ( (out_args->status = RSTSL_Finish( )) != E_SUCCESS)
482 1                  status  = COMMAND_RESULT_FAILURE;

484 1          if (NULL != input_args)
485 2          {
486 2                  xdr_free( xdr_RE_null_args, (char *)input_args);
487 2                  free( input_args );
488 2                  *output_args = (void *)out_args;
489 1          }
490 1          else    /* only keep output struct if user want it */
491 1                  free( out_args );

493 1          return status;
494  }
```



```
496  /*****************************************************************************
                                                                          ********
497  **
498  ** Routine:  EDMRE_FindRestorableObjects
499  **
500  ** Inputs:   void *input_args    ptr to struct with RPC input args
501  **
502  ** Outputs:  void **status       addr of void * to receive ptr output
                                                                        arg struct
503  **
504  ** Return Codes:
505  **           0 for success and non-zero for failure.
506  **
507  ** Purpose:  Wrapper of Restore service library call to be executed
508  **            asynchronously from main RPC thread
509  **
510  *****************************************************************************
                                                                          ********
511  */
512  int EDMRE_FindRestorableObjects(
                                    void *input_args, void **output_args )
513 1 {
514 1          RE_find_restorable_objects_args         *in_args =
515 1                  (RE_find_restorable_objects_args *)input_args;
516 1          RE_find_restorable_objects_result       *out_args;
517 1          EBREC_SearchCriteriaRec
                                                            searchCriteria;

519 1          int     status = COMMAND_RESULT_SUCCESS;

521 1          out_args = calloc( 1, sizeof(
                                    RE_find_restorable_objects_result) );
522 1          if (NULL == out_args)
523 2          {
524 2                  EDMRestoreEng_logent( __FILE__, __LINE__, LOG_ERR,
525 2                                  MESSAGE_NO_MEMORY, 0,
526 2                          "calloc fail for
                                    RE_find_restorable_objects_result" );
527 2                  status = COMMAND_RESULT_FAILURE;
                                                        /* fatal error */
528 1          }

530 1          else
531 2          {
532 2                  /* prepare search criteria structure for input to
                                                            RSTST func: */
533 2                  strncpy( searchCriteria.startDirectory,
534 2                          in_args->searchCriteria->startDirectory,
                                                                    256);
535 2                  searchCriteria.descendDirectory =
536 2
                            in_args->searchCriteria->descendDirectory;
537 2                  strncpy( searchCriteria.searchString,
538 2                          in_args->searchCriteria->searchString, 128);
539 2                  searchCriteria.excludeString =
540 2
                                in_args->searchCriteria->excludeString;
541 2                  searchCriteria.typeOfFile =
542 2                              in_args->searchCriteria->typeOfFile;
543 2                  strncpy( searchCriteria.owner,
544 2                          in_args->searchCriteria->owner, 64);
545 2                  searchCriteria.excludeOwner =
546 2                              in_args->searchCriteria->excludeOwner;
547 2                  strncpy( searchCriteria.group,
548 2                          in_args->searchCriteria->group, 64);
```

```
549 2                   searchCriteria.excludeGroup =
550 2                                   in_args->searchCriteria->excludeGroup;
551 2                   searchCriteria.sizeInBytes.high =
552 2                                   in_args->searchCriteria->sizeInBytes.
                                                                        high;
553 2                   searchCriteria.sizeInBytes.low =
554 2                                   in_args->searchCriteria->sizeInBytes.
                                                                        low;
555 2                   searchCriteria.sizeMatch =
556 2                                   in_args->searchCriteria->sizeMatch;
557 2                   searchCriteria.startTime =
558 2                                   in_args->searchCriteria->startTime;
559 2                   searchCriteria.endTime =
560 2                                   in_args->searchCriteria->endTime;

562 2                   out_args->status = RSTSL_FindRestorableObjects(
563 2
                                                        &searchCriteria,
564 2
                                                    EDMRE_ProgressCallback );
565 2                   *output_args = (void *)out_args;
566 1           }

568 1           xdr_free( xdr_RE_find_restorable_objects_args, (
                                                        char *)in_args );
569 1           free( in_args );

571 1           return status;
572       }
```



```
576   /*****************************************************************************
                                                                          ********
577   **
578   ** Routine:  int EDMRE_Load_recx_directives
579   **
580   ** Inputs:  RE_recx_file_info *fileinfo   Information on file to be
                                                                        retreived
581   **
582   ** Outputs: Error or success from the RSTSL call
583   **
584   ** Return Codes:
585   **           0 for success and non-zero for failure.
586   **
587   ** Purpose:  Function to retrieve directives file from client and then
588   **           load the file contents into the context structure.
                                                                        The file
589   **           transfer is done with edm link.
590   **
591   *****************************************************************************
                                                                          ********
592   */
593   int EDMRE_Load_recx_directives( void *input_args,
594                                   void **output_args )
595 1 {
596 1   RSTRPC_recx_file_info * fileinfo = (
                                        RSTRPC_recx_file_info *)input_args;
597 1   RE_status_result *outargs;
598 1   outargs = calloc(1,sizeof(RE_status_result));

600 1   /*
601 1    * Actually load the recx structure.
602 1    */
603 1   outargs->status = RSTSL_Load_recx_directives(fileinfo);
604 1   *output_args = (void *)outargs;

606 1   /*
607 1    * Return that the RPC was atleast successfule,
                                                    the load may not have been
608 1    */
609 1   return COMMAND_RESULT_SUCCESS;

611   }
```

```
614  /*****************************************************************************
                                                                          ********
615  **
616  ** Routine:  EDMRE_SetPreviousBackup
617  **
618  ** Inputs:   void *input_args    ptr to struct with RPC input args
619  **
620  ** Outputs:  void **status       addr of void * to receive ptr output
                                                                    arg struct
621  **
622  ** Return Codes:
623  **           0 for success and non-zero for failure.
624  **
625  ** Purpose:  Wrapper of Restore service library call to be executed
626  **           asynchronously from main RPC thread
627  **
628  *****************************************************************************
                                                                          ********
629  */
630  int EDMRE_SetPreviousBackup( void *input_args, void **output_args )
631 1{
632 1    RE_set_backup_time_args       *in_args
633 1         = (RE_set_backup_time_args *)input_args;
634 1    RE_get_all_backup_times_result       *out_args;

636 1    int           status = COMMAND_RESULT_SUCCESS;

638 1    out_args = calloc( 1, sizeof (RE_status_result) );


641 1    if (NULL == out_args)
642 2    {
643 2         EDMRestoreEng_logent(
                        __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
644 2                  0, "calloc fail for RE_status_result" );
645 2         status = COMMAND_RESULT_FAILURE;        /* fatal error */
646 1    }

648 1    else
649 2    {
650 2         out_args->status = RSTSL_SetPrevBackup( in_args->flags);
651 2         *output_args = (void *) out_args;
652 1    }

654 1    xdr_free( xdr_RE_set_backup_time_args, (char *)in_args);
655 1    free( in_args);

657 1    return status;
658  }
```



```
662  /*****************************************************************************
                                                                          ********
663  **
664  ** Routine:  EDMRE_SetBackupForTime
665  **
666  ** Inputs:   void *input_args    ptr to struct with RPC input args
667  **
668  ** Outputs:  void **status       addr of void * to receive ptr output
                                                                    arg struct
669  **
670  ** Return Codes:
671  **           0 for success and non-zero for failure.
672  **
673  ** Purpose:  Wrapper of Restore service library call to be executed
674  **           asynchronously from main RPC thread
675  **
676  *****************************************************************************
                                                                          ********
677  */
678  int EDMRE_SetBackupForTime( void *input_args, void **output_args )
679 1{
680 1    RE_backup_for_time_args       *in_args
681 1         = (RE_backup_for_time_args *)input_args;
682 1    RE_get_all_backup_times_result       *out_args;

684 1    int           status = COMMAND_RESULT_SUCCESS;

686 1    out_args = calloc( 1, sizeof (RE_status_result) );


689 1    if (NULL == out_args)
690 2    {
691 2         EDMRestoreEng_logent(
                        __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
692 2                  0, "calloc fail for RE_status_result" );
693 2         status = COMMAND_RESULT_FAILURE;        /* fatal error */
694 1    }
695 1    else
696 2    {
697 2         out_args->status = RSTSL_SetBackupForTime( in_args->time,
698 2                                                    in_args->flags);
699 2         *output_args = (void *) out_args;
700 1    }

702 1    xdr_free( xdr_RE_backup_for_time_args, (char *)in_args);
703 1    free( in_args);

705 1    return status;
706  }
```

```
708  /*****************************************************************************
                                                                          ********
709  **
710  ** Routine:  EDMRE_SetNextBackup
711  **
712  ** Inputs:   void *input_args    ptr to struct with RPC input args
713  **
714  ** Outputs:  void **status       addr of void * to receive ptr output
                                                                    arg struct
715  **
716  ** Return Codes:
717  **           0 for success and non-zero for failure.
718  **
719  ** Purpose:  Wrapper of Restore service library call to be executed
720  **            asynchronously from main RPC thread
721  **
722  *****************************************************************************
                                                                          ********
723  */
724  int EDMRE_SetNextBackup( void *input_args, void **output_args )
725 1 {
726 1     RE_set_backup_time_args      *in_args
727 1          = (RE_set_backup_time_args *)input_args;
728 1     RE_get_all_backup_times_result       *out_args;

730 1     int          status = COMMAND_RESULT_SUCCESS;

732 1     out_args = calloc( 1, sizeof (RE_status_result) );


735 1     if (NULL == out_args)
736 2     {
737 2          EDMRestoreEng_logent(
                        __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
738 2                  0, "calloc fail for RE_status_result" );
739 2          status = COMMAND_RESULT_FAILURE;        /* fatal error */
740 1     }

742 1     else
743 2     {
744 2          out_args->status = RSTSL_SetNextBackup( in_args->flags);
745 2          *output_args = (void *) out_args;
746 1     }

748 1     xdr_free( xdr_RE_set_backup_time_args, (char *)in_args);
749 1     free( in_args);

751 1     return status;
752  }
```



```
754  /*****************************************************************************
                                                                          ********
755  **
756  ** Routine:  EDMRE_SetFirstBackup
757  **
758  ** Inputs:   void *input_args    ptr to struct with RPC input args
759  **
760  ** Outputs:  void **status       addr of void * to receive ptr output
                                                                    arg struct
761  **
762  ** Return Codes:
763  **           0 for success and non-zero for failure.
764  **
765  ** Purpose:  Wrapper of Restore service library call to be executed
766  **            asynchronously from main RPC thread
767  **
768  *****************************************************************************
                                                                          ********
769  */
770  int EDMRE_SetFirstBackup( void *input_args, void **output_args )
771 1 {
772 1     RE_set_backup_time_args      *in_args
773 1          = (RE_set_backup_time_args *)input_args;
774 1     RE_get_all_backup_times_result       *out_args;

776 1     int          status = COMMAND_RESULT_SUCCESS;

778 1     out_args = calloc( 1, sizeof (RE_status_result) );


781 1     if (NULL == out_args)
782 2     {
783 2          EDMRestoreEng_logent(
                        __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
784 2                  0, "calloc fail for RE_status_result" );
785 2          status = COMMAND_RESULT_FAILURE;        /* fatal error */
786 1     }

788 1     else
789 2     {
790 2          out_args->status = RSTSL_SetFirstBackup( in_args->flags);
791 2          *output_args = (void *) out_args;
792 1     }

794 1     xdr_free( xdr_RE_set_backup_time_args, (char *)in_args);
795 1     free( in_args);

797 1     return status;
798  }
```

```
800   /*********************************************************************
                                                                      ********
801   **
802   ** Routine:  EDMRE_SetMostRecentBackup
803   **
804   ** Inputs:   void *input_args     ptr to struct with RPC input args
805   **
806   ** Outputs:  void **status       addr of void * to receive ptr output
                                                                    arg struct
807   **
808   ** Return Codes:
809   **           0 for success and non-zero for failure.
810   **
811   ** Purpose:  Wrapper of Restore service library call to be executed
812   **            asynchronously from main RPC thread
813   **
814   *********************************************************************
                                                                      ********
815   */
816   int EDMRE_SetMostRecentBackup( void *input_args, void **output_args )
817 1 {
818 1     RE_set_backup_time_args      *in_args
819 1          = (RE_set_backup_time_args *)input_args;
820 1     RE_get_all_backup_times_result       *out_args;

822 1     int          status = COMMAND_RESULT_SUCCESS;

824 1     out_args = calloc( 1, sizeof (RE_status_result) );


827 1     if (NULL == out_args)
828 2     {
829 2          EDMRestoreEng_logent(
                        __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_MEMORY,
830 2                  0, "calloc fail for RE_status_result" );
831 2          status = COMMAND_RESULT_FAILURE;        /* fatal error */
832 1     }

834 1     else
835 2     {
836 2          out_args->status = RSTSL_SetMostRecentBackup( in_args->flags);
837 2          *output_args = (void *) out_args;
838 1     }

840 1     xdr_free( xdr_RE_set_backup_time_args, (char *)in_args);
841 1     free( in_args);

843 1     return status;
844   }
```

```
/*
** ========================================================================
                                                                    ========
** Copyright 1996,1997 EMC Corporation
** ========================================================================
                                                                    ========
*/
/*
** ========================================================================
                                                                    ========
** EDMFinalStatus.c
**
** Mission Statement:
**
**
**
**
** Primary Data Acted On:
**
** Compile-Time Options:
**
**                      USE_SUNRPC - Compile source with sunrpc
                                                          support.  If
**                                  not set, assume DCE support.
**
** Basic idea here: Module for Control Channel Reader thread.
** ========================================================================
                                                                    ========
*/
/*
** ========================================================================
                                                                    ========
** The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
** ========================================================================
                                                                    ========
*/
#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMFinalStatus.c,v $ "
                            "$Revision: 1.23 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

/* #define _POSIX_SOURCE   unable to compile with this define set */
/* #define _XOPEN_SOURCE   unable to compile with this define set */

#include <sys/types.h>
#include <sys/utsname.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/inout.h>

#include <csc/cscomm.h>
#include <pthread.h>

// Rogue Wave includes
#include <rw/collect.h>
#include <rw/rwfile.h>
#include <rw/vstream.h>
#include <rw/bintree.h>
```



```
#ifdef __cplusplus
extern "C" {
#endif

#include <EDMutils.h>
#include <restore/dispatch_daemon.h>
#include <restore/csc_Dispatch_Protocol_Client.h>
#include <restore/dispatch_protocol_client.h>
#include <restore/dispatch_protocol_service.h>
#include <restore/dispatch_protocol.h>
#include <EDMFinalStatus.h>

#ifdef __cplusplus
}
#endif
#include <EDMRE_ccr.h>
#include <EDMSession.h>
#include <logging/logging.h>
#include <EDMReturnMessageApi.h>
#include <EDMRestoreEngLog.h>

// Global/Extern if_spec to be used by RE_ccw/DD_ccr.
rpc_binding_handle_t *restoreService_ccw_handle_p;
extern DD_client_session_id *p_restoreServiceUid;

int
SendFinalStatus(void)
{
 int lrc=0;
 int status=0;

         lrc = PushResponseMessage( (int) dp_final_stats_indicate,
                                    *p_restoreServiceUid,
                                    restoreService_ccw_handle_p,
                                    &status );

 return(lrc);
}
```

```
/*****************************************************************************
                                                                          ******
**
** File Name:   RSLinitfin.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
**               This module contains the Restore Service Library
                                                                  functions to
**               initialize and terminate the restore operation.
**
** Table of Contents:
** ------------------
**               RSTSL_Initialize
**               RSTSL_Finish
**
**       Internal Functions:
**
**
** Compile-Time Options:
**               This section must list any compile time definitions
**               which will affect this header.
**
*****************************************************************************
                                                                      *****/



/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif


/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define  _POSIX_SOURCE 1


/*
 * System headers.
 */
#include <sys/param.h>
#include <dirent.h>
#include <dlfcn.h>


/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>
```



```
/*
 * Local headers
 */
#include <RSLinterns.h>
#include <RSLstartup.h>


/*
 * #defines, structures, typedefs local to this source file
 */

static eerrno_ty init_plugins( restore_context *rcp );
static int validate_plugin( struct pluginData *piDataPtr );


/*
 * External declarations
 *
 * This is the global "restore context" that will be used
 * throughout the rest of the restore operations.
 */
struct restore_context *rcp = NULL;


/*
 * Definitions of the names of the plugin functions in the piFuncArray
 * of the pluginData structure.
                                These must be in the same order and position
 * as the PIFuncIndex values defined in RSLplugin.h.
 */

char *piFuncNames[PIFuncIndexLast+1] = {
        "RSTPI_Identify",
        "RSTPI_Initialize",
        "RSTPI_GetTopLevelObjects",
        "RSTPI_SetTopLevelObject",
        "RSTPI_GetNextLevelObjects",
        "RSTPI_ClearRestoreContext",
        "RSTPI_Submit",
        "RSTPI_GetTopLevelTemplates",
        "RSTPI_DoesAlternateExist",
        "RSTPI_MarkObject",
        "RSTPI_UnmarkObject",
        "RSTPI_IsObjectMarked",
        "RSTPI_IsObjectMarkable",
        "RSTPI_GetAllBackupTimes",
        "RSTPI_GetCurrentBackupTime",
        "RSTPI_SetBackupForTime",
        "RSTPI_SetPrevBackup",
        "RSTPI_SetNextBackup",
        "RSTPI_SetFirstBackup",
        "RSTPI_SetMostRecentBackup",
        "RSTPI_IsTherePrevBackup",
        "RSTPI_IsThereNextBackup",
        "RSTPI_IsTherePrevBackupForTime",
        "RSTPI_IsThereNextBackupForTime",
        "RSTPI_Finish",
        "RSTPI_StartRestore",
        "RSTPI_FindRestorableObjects",
        "RSTPI_GetFindResults",
        "RSTPI_GetNecessaryMedia"
};
```

```
127  /*****************************************************************
128   * RSTSL_Initialize:
129   *
130   *  This function takes care of all the initialization for a restore
131   *  session.  This must be called prior to any of the other functions
132   *  in the Restore API.
133   *
134   * Parameters:
135   *
136   *  userName (I) - The name of the user.
137   *
138   *****************************************************************/

140  eerrno_ty
141  RSTSL_Initialize( const char *userName )
142 1 {
143 1     eerrno_ty status = E_SUCCESS;

145 1     /*
146 1      * If we have not yet allocated space for a restore_context
147 1      * structure, do so now.  If we have already done so,
                                                        just clear it
148 1      * now.
149 1      */

151 1     if (NULL == rcp)
152 2     {
153 2         rcp = (struct restore_context *)malloc(sizeof(
                                            struct restore_context));
154 2         if (NULL == rcp)
155 3          {
156 3              rec_api_log_csm(SUB_CSM_NOMEM, NULL);
157 3              return(EP_RB_RECOVER_NOMEM);
158 2          }
159 1     }
160 1     memset(rcp, 0, sizeof(struct restore_context));

162 1     rcp->rc_human_uidname = esl_strdup( userName );


165 2     if (!rcp->rc_human_uidname) {
166 2       rec_api_log_csm(SUB_CSM_NOMEM, NULL);
167 2       return(EP_RB_RECOVER_NOMEM);
168 1     }

170 1     /*
171 1      * Set the appropriate field in the recovery context to indicate
172 1      * that this recover session is based on the Recover API.
173 1      * This flag is in place for historical reasons but is used by
174 1      * other parts of the Recover API library.
175 1      */

177 1     rcp->gui_mode = 1;

179 1     /*
180 1      * Initialize the logging mechanism.
181 1      */

183 1     if (status = rbrlog_begin(rcp, progname))
184 2     {
185 2          return(status);
186 1     }

188 1     /*
189 1      * Initialize the few "recover context" variables that we can at
190 1      * this early stage.
```



```
191 1      */

193 1     setup_proc(rcp);

195 1     /*
196 1      * The following call will:
197 1      *   -Initialize the saveset datatbase.
198 1      *   -Infer any information we can at this point.
199 1      */

201 1     if (status = startup(rcp))
202 2     {
203 2         return(status);
204 1     }

206 1     /* Do plugins setup: Find and initialize all valid restore plugin
                                                              libs: */

208 1     status = init_plugins( rcp );

210 1     return( status );
211  }         /* End of RSTSL_Initialize() */
```

```
214  /*****************************************************************
215   * RSTSL_Finish
216   *
217   *    Function Description:
218   *
219   *  This function terminates a restoral session,
                                          but not while a restore is in
220   *  progress.
           It will be rejected if a restore is currently being executed.
221   *  This routine will clean up any local memory used in the session.
222   *
223   * Parameters:
224   *
225   *  none
226   *
227   */

229  eerrno_ty
230  RSTSL_Finish( void )
231 1 {
232 1     int mc_n;


235 1     eerrno_ty err = E_SUCCESS;

237 1     if (NULL == rcp)
238 2     {
239 2          return( E_SUCCESS );
240 1     }
241 1     RemoveSubmitFiles();
242 1     /*
243 1      * Call rbr_cleanup() which kills the aux proc(
                                                    s), unlocks the work
244 1      * item, then calls rbrlog_end(
                                       ) to enter the last logs and to close
245 1      * the log file.
246 1      */

248 1     rbr_cleanup(rcp);

250 1     /*
251 1      * Deallocate the memory of restore_context and the related
252 1      * structures.
253 1      */

255 1     if (NULL != rcp->rc_mcp)      /* Free the multicat structures */
256 2     {
257 2          mcat_destroy(rcp->rc_mcp);
258 1     }

260 1     /*
261 1      * Free the mark bit map space
262 1      */

264 1     for (mc_n = 0; mc_n < rcp->rc_marks_plane_alloc; mc_n++)
265 2     {
266 2          if (NULL != rcp->rc_marks[mc_n])
267 3          {
268 3              free(rcp->rc_marks[mc_n]);
269 2          }
270 2          rcp->rc_marks[mc_n] = NULL;
271 1     }

273 1     if (NULL != rcp->rc_nmarks_by_plane)
```



```
274 2     {
275 2          free(rcp->rc_nmarks_by_plane);
276 1     }

278 1     /*
279 1      * Free the configuration structures
280 1      */

282 1 #if 0
283 1     if (NULL != rcp->rc_cfgname)
284 2     {
285 2          free(rcp->rc_cfgname);
286 1     }
287 1 #endif

289 1     if (NULL != rcp->rc_config)
290 2     {
291 2          rbc_freeconfig(rcp->rc_config);
292 1     }

294 1     /*
295 1      * Free the DS_NONE structures array
296 1      * Note that even though rc_dsnones is the head of linked list
297 1      * of dsnone_info structures, the list is allocated via malloc
298 1      * as an array initially (ref. alloc_plane_arrays()), therefore
299 1      * we can do a free here.
300 1      */

302 1     if (NULL != rcp->rc_dsnones)
303 2     {
304 2          free(rcp->rc_dsnones);
305 1     }

307 1     /*
308 1      * Free the volume list structures.
309 1      */

311 1     if (NULL != rcp->ebvllist)
312 2     {
313 2          (void)ebvl_volidlist_destructor(
                                        rcp->ebvllist, EBVL_DESTROY_ALL);
314 1     }

316 1     /*
317 1      * Free the plugin related data
318 1      */

320 1     rcp->rc_backup_app = 0;
321 1     while (rcp->currentPIptr = rcp->piList)
322 2     {
323 2          rcp->rc_backup_app++;
324 2          rcp->appData = rcp->currentPIptr->appData;
325 2          /* allow plugin to clean up and close .so: */
326 2          if ( E_SUCCESS != (err =
327 2               rcp-> currentPIptr-> piFuncArray[ PIFuncIndexFinish ] (
                                                              rcp) ) )
328 3          {
329 3          /* log error, continue */
330 3              rbe_user_error( err,
331 3                          "RSTPI_Finish failed for restore plug-in
                                                          library %s\n",
332 3                  ((struct pluginIDdata *)(
                                  rcp-> currentPIptr-> idData))-> name );
333 2          }
334 2          dlclose( rcp-> currentPIptr-> libHdl );
335 2          rcp->piList = rcp->piList->next;
```

```
336 2           free (rcp->currentPIptr);
337 1       }

339 1       /*
340 1        * Free the various simple string buffers
341 1        */

343 1       if (NULL != rcp->rc_top_level_object_name)
344 2       {
345 2            free(rcp->rc_top_level_object_name);
346 1       }

348 1       if (NULL != rcp->rc_template_name)
349 2       {
350 2            free(rcp->rc_template_name);
351 1       }

353 1       if (NULL != rcp->rc_workitem_name)
354 2       {
355 2            free(rcp->rc_workitem_name);
356 1       }

358 1       if (NULL != rcp->rc_human_uidname)
359 2       {
360 2            free(rcp->rc_human_uidname);
361 1       }

363 1       if (NULL != rcp->rc_effective_uidname)
364 2       {
365 2            /* don't free, its internal: free(rcp->rc_effective_uidname);
                                                                        */
366 1       }

368 1       if (NULL != rcp->rc_client_rbuname)
369 2       {
370 2            free(rcp->rc_client_rbuname);
371 1       }

373 1       if (NULL != rcp->rc_client_hostname)
374 2       {
375 2            free(rcp->rc_client_hostname);
376 1       }

378 1       if (NULL != rcp->rc_client_scriptname)
379 2       {
380 2            /* don't free, its internal: free(rcp->rc_client_scriptname);
                                                                        */
381 1       }

383 1       if (NULL != rcp->rc_client_dirtop)
384 2       {
385 2            free(rcp->rc_client_dirtop);
386 1       }

388 1       if (NULL != rcp->rc_cmd_context)
389 2       {
390 2            /* don't free -- its internal/temp data: free(
                                                rcp->rc_cmd_context); */
391 1       }

393 1       if (NULL != rcp->rc_source_client_hostname)
394 2       {
395 2            free(rcp->rc_source_client_hostname);
396 1       }

398 1       if (NULL != rcp->rc_cpiogen_executable)
```



```
399 2       {
400 2            /* don't free, its internal: free(rcp->rc_cpiogen_executable);
                                                                        */
401 1       }

403 1       if (NULL != rcp->rc_plugin_wi_types)
404 2       {
405 2            free(rcp->rc_plugin_wi_types);
406 1       }

408 1       if (NULL != rcp->rc_pwd)
409 2       {
410 2            free(rcp->rc_pwd);
411 1       }

413 1       /*
414 1        * Finally, deallocate the restore_context itself
415 1        */

417 1       free(rcp);
418 1       rcp = NULL;



422 1       return( err );
423     }        /* RSTSL_Finish */
```

```
427  /*****************************************************************
428   * init_plugins
429   *
430   *    Function Description:
431   *
432   *  This function locates, opens,
                                validates and initializes all restore
433   *  plug-in (shared) libraries.  They must be located in
434   *  /usr/epoch/EB/cure_plugin (
                            eb_cure_plugin_dir).  All .so files in that
435   *  directory are opened and validates for version# and presence of
                                                                  all
436   *  mandatory functions.
                         The RSTPI_Identify function is called for each
437   *  library to determine which optional features are supported,
                                                              and that
438   *  the corresponding functions are present.  Finally,
                                                    the RSTPI_Initialize
439   *  function is called for each valid library.
440   *
441   * Parameters:
442   *
443   * Inputs:
444   *       rcp     (I)     - Pointer to restore context
445   *
446   * Outputs:
447   *       none
448   *
449   * Returns:
450   *       E_SUCCESS or EP_RB_RECOVER_xxx
451   *
452   * Logic/pseudo code:
453   *
454   *       open plugin dir
455   *       while read_next_entry succeeds
456   *               verify .so file (else continue)
457   *               open shared library file (else continue)
458   *               on errors below:
459   *                       close shared library file
460   *                       continue
461   *               fetch all mandatory function addresses
462   *               call Identify function
463   *               validate version number
464   *               fetch all indicated optional function addrs
465   *               call Initialize function
466   *               add workitem types to composite exclusion list
467   *               add to valid plugin list
468   *       close plugin dir
469   */

471  static eerrno_ty init_plugins( restore_context *rcp )
472  {
473          DIR             *dirp;
474          struct dirent   *direntp;
475          eerrno_ty       status = E_SUCCESS;
476          struct pluginData *piDataPtr = NULL;
477          struct pluginData *piListPtr = NULL;
478          int             val_result;
479          struct pluginIDdata *idDataPtr;
480          char            *tmp_types;
481          int             shlib_dirlen;
482          char            shlib_path [MAXPATHLEN];
```



```
484  1          /* open plugin directory or bust */
485  1          if ( NULL == (dirp = opendir( eb_cure_plugin_dir )) )
486  2          {
487  2                  rec_api_log_csm( SUB_CSM_PLUGIN_ERR, NULL );
488  2  #if 1
489  2                  return E_SUCCESS;
                                    /* allow continuation w/o plugins */
490  2  #else
491  2                  return EP_RB_RECOVER_NO_PLUGINS;
                                                    /* later do this */
492  2  #endif
493  1          }

495  1          strcpy( shlib_path, eb_cure_plugin_dir );
496  1          strcat( shlib_path, "/" );
497  1          shlib_dirlen = strlen (shlib_path);

499  1          /* loop thru entries in directory*/
500  1          while (NULL != (direntp = readdir( dirp )))
501  2          {
502  2                  if (NULL == piDataPtr)
503  3                  {       /* allocate next plugin data structure */
504  3                          if (NULL == (piDataPtr
505  3                                  = calloc( 1, sizeof(
                                                struct pluginData) )))
506  4                          {
507  4                                  status = EP_RB_RECOVER_NOMEM;
508  4                                  break;
                                                /* fall thru to cleanup */
509  3                          }
510  2                  }

512  2                  if (NULL == strstr( direntp->d_name, ".so" ) )
513  2                          continue;       /* skip this guy */

515  2                  strcpy( &shlib_path[shlib_dirlen], direntp->d_name );
516  2                  if (NULL == (piDataPtr->libHdl
517  2                           = dlopen( shlib_path, RTLD_NOW )))
518  3                  {
519  3                          rbe_user_error( 0,
520  3                              "Error opening restore plug-in library
                                                          %s: %s\n",
521  3                                      direntp->d_name, dlerror() );
522  3                          continue;       /* skip this one */
523  2                  }

525  2          /* Fetch addresses of all mandatory functions and */
526  2          /* Do Identify processing: call it, save options, validate */

528  2                  if (0 != (val_result = validate_plugin(
                                                      piDataPtr ) ))
529  3                  {
530  3                      if (val_result == -1 || val_result == -4)
531  4                      {
532  4                          rbe_user_error( 0,
533  4                  "Functions missing from restore plug-in library %s:
                                                          %s\n",
534  4                                      direntp->d_name, dlerror() );
535  3                      }
536  3                      else if (val_result < 0)
537  4                      {
538  4                          rbe_user_error( 0,
539  4                            "Validation failed for restore plug-in
                                                      library %s\n",
540  4                                      direntp->d_name );
541  3                      }
```

```
542 3                       else
543 4                       {
544 4                           rbe_user_error( val_result,
545 4                   "RSTPI_Identify failed for restore plug-in library
                                                                %s\n",
                                        direntp->d_name );
546 4                       }
547 3
549 3                       dlclose( piDataPtr->libHdl );
                                        /* close .so on errors */
550 3                       piDataPtr->libHdl = NULL;
551 3                       continue;   /* on any error, skip this lib */
552 2                   }

554 2           /* let DC plug-in do its initialization */
555 2                   rcp->appData = NULL;
                                /* enter plugin with clean appdata */
556 2                   status =
                       piDataPtr->piFuncArray[PIFuncIndexInitialize]( rcp );
557 2                   if (E_SUCCESS != status)
558 3                   {
559 3                           rbe_user_error( status,
560 3                   "RSTPI_Initialize failed for restore plug-in library
                                                                %s\n",
                                        direntp->d_name );
561 3                           dlclose( piDataPtr->libHdl );
562 3                                   /* close .so on errors */
563 3                           piDataPtr->libHdl = NULL;
564 3                           status = E_SUCCESS;
                                           /* this was't fatal */
565 3                           continue;
                                   /* on any error, skip this lib */
566 2                   }

568 2           /* save plugin's appData */
569 2                   piDataPtr->appData = rcp->appData;
570 2                   rcp->appData = NULL;

572 2           /* add piDataPtr to valid plugin list */
573 2                   if (NULL == piListPtr)
574 2                           rcp->piList= piDataPtr;
                                           /* first in list */
575 2                   else
576 2                           piListPtr->next = piDataPtr;
                                           /* link from prev */
577 2                   piListPtr = piDataPtr;
                                           /* new end of list */
578 2                   piDataPtr = NULL;

580 2           /* add workitem types to composite exclusion list */
581 2                   idDataPtr = (struct pluginIDdata *)piListPtr->idData;
582 2                   if (idDataPtr->num_types > 0)
583 3                   {
584 3                       tmp_types = calloc( 1, 1 + idDataPtr->num_types
585 3                                   + rcp->rc_num_plugin_wi_types
                                                                    );
586 4                       if (NULL == tmp_types) {
587 4                           status = EP_RB_RECOVER_NOMEM;
588 4                           break;
589 3                       }
590 3                       if (NULL != rcp->rc_plugin_wi_types)
591 4                       {
                          /* move old list to new buffer and free old list */
592 4                           memcpy( tmp_types,
593 4                                   rcp->rc_plugin_wi_types,
594 4                                   rcp->rc_num_plugin_wi_types );
```



```
595 4                           free( rcp->rc_plugin_wi_types );
596 3                       }
597 3                       memcpy( tmp_types + rcp->rc_num_plugin_wi_types,
598 3                               idDataPtr->wi_types,
599 3                               idDataPtr->num_types );
600 3                       rcp->rc_num_plugin_wi_types +=
                                                    idDataPtr->num_types;
601 3                       tmp_types[rcp->rc_num_plugin_wi_types] = 0;
602 3                       rcp->rc_plugin_wi_types = tmp_types;
603 2                   }
604 1           }

606 1           (void)closedir( dirp );

608 1           /* free up leftovers: */
609 1           if (NULL != piDataPtr)
610 1                   free (piDataPtr);

612 1           if (E_SUCCESS != status)
613 2           { /* Free contents of plugin list: */
614 2                   while (NULL != (piDataPtr = piListPtr))
615 3                   { /* allow plugin to clean up and close .so: */
616 3                           rcp->appData = piDataPtr->appData;
617 3                           piDataPtr->piFuncArray[PIFuncIndexFinish](
                                                                    rcp );
618 3                           dlclose( piDataPtr->libHdl );
619 3                           piListPtr = piDataPtr->next;
620 3                           free (piDataPtr);
621 2                   }
622 1           }

624 1           return status;
625   }
```

```
626         /* init_plugins */

628 /*****************************************************************
629  * validate_plugin
630  *
631  *    Function Description:
632  *
633  *  This function retrieves the addresses of the mandatory plugin
                                                              functions
634  *  and stores them in the function pointer array.
                                                  If any function is missing
635  *  it returns -1.
           It then calls the identify function and verifies wthe plugin
636  *  version,
            and finds its optional functions. Specific error values are
637  *  returned on version mismatch and missing optional functions.
638  *
639  * Parameters:
640  *
641  * Inputs:
642  *       piDataPtr (
                I)   - Pointer to plugin data structure with libHdl set
643  *
644  * Outputs:
645  *       piFuncArray in piDataPtr is loaded with pointers to plugin
                                                              functions
646  *
647  * Returns:
648  *       0 on success
649  *       -1 on any missing required functions
650  *       -2 if version validation fails OR identify returns junk
651  *       -3 if workitem type validation fails
652  *       -4 on any missing optional functions indicated by options
                                                                  flags
653  *       +n (
   EB_RB_RECOVER_xxx) for error codes returned from Identify function
654  *
655  *****************************************************************
                                                                  ****/


657 static int validate_plugin( struct pluginData *piDataPtr )
658 1 {
659 1         int             index;
660 1         eerrno_ty       status;
661 1         struct pluginIDdata *idDataPtr;

663 1         for( index = 0; index <= PIFuncIndexLastBasic; index++ )
664 2         {
665 2                 if (NULL == (piDataPtr->piFuncArray[index]
666 2                         = (piFuncPtr) dlsym( piDataPtr->libHdl,
667 2                                                 piFuncNames[index]
                                                                    ))

668 2                 )
669 2                         return -1;
670 1         }

672 1         /* call identify and validate: */
673 1         status = piDataPtr->piFuncArray[PIFuncIndexIdentify](
674 1                                                 &piDataPtr->idData );

675 1         if (status != E_SUCCESS)
676 1                 return status;
677 1         if (NULL == (idDataPtr = (
                                struct pluginIDdata *)piDataPtr->idData) )
678 1                 return -2;
```



```
680 1         if (idDataPtr->version != RSTPI_VERSION)
681 2         { /* only version 1 supported so far */
682 2                 piDataPtr->idData = NULL;
683 2                 return -2;
684 1         }

686 1         if (idDataPtr->num_types && !idDataPtr->wi_types)
687 2        {        /* count cant be positive with null pointer */
688 2                piDataPtr->idData = NULL;
689 2                return -3;
690 1        }

692 1         /* if startRestore option set, get its addr or bust */
693 1         if ( ( (RSTPI_OPTION_SPECIAL_START
694 1                 == (idDataPtr->options & RSTPI_OPTION_MASK_START))
695 1             && (NULL == (
                        piDataPtr->piFuncArray[PIFuncIndexStartRestore]
696 1                        = (piFuncPtr) dlsym( piDataPtr->libHdl,
697 1                                                piFuncNames[PIFuncIndexStartRestore] )))
698 1           )
699 1         /* OR if special find option set, get its addr or bust */
700 1           || ( (RSTPI_OPTION_SPECIAL_FIND
701 1                 == (idDataPtr->options & RSTPI_OPTION_MASK_FIND))
702 1             && ( (NULL == (piDataPtr->piFuncArray[PIFuncIndexFind]
703 1                        = (piFuncPtr) dlsym( piDataPtr->libHdl,
704 1                                                piFuncNames[PIFuncIndexFind] )))
705 1               || (NULL == (
                        piDataPtr->piFuncArray[PIFuncIndexFindResults]
706 1                        = (piFuncPtr) dlsym( piDataPtr->libHdl,
707 1                                                piFuncNames[PIFuncIndexFindResults])
708 1               ) ) )         )
709 1         /* OR if special getMedia option set, get its addr or bust */
710 1           || ( (RSTPI_OPTION_SPECIAL_GET_MEDIA
711 1                 == (
                        idDataPtr->options & RSTPI_OPTION_MASK_GET_MEDIA))
712 1             && (NULL == (piDataPtr->piFuncArray[PIFuncIndexGetMedia]
713 1                        = (piFuncPtr) dlsym( piDataPtr->libHdl,
714 1                                                piFuncNames[PIFuncIndexGetMedia] )))
715 1         )     )
716 2         {
717 2                 piDataPtr->idData = NULL;
718 2                 return -4;
719 1         }

721 1         return 0;
722 }
```

```
723  /*       validate_plugin */
```

```
1   /*
2   ** Copyright 1996,1997 EMC Corporation
3   */



6   /* EDMReturnMessageApi.cc
7    *
8    *
9    * Mission Statement:  file that contains an API to manage the Message
                                                                    Queues
10   *
11   * Primary Data Acted On:
12   *
13   * Compile-Time Options:
14   *
15   * Basic idea here:
16   *                     A few calls to manage the Message Queues.
17  */

19  #if !defined(lint)
20  static char     RCS_id [] = "@(
                                 #)$RCSfile: EDMReturnMessageApi.cc,v $ "
21                              "$Revision: 1.0 $ "
22                              "$Date: 1997/02/06 20:49:15 $" ;
23  #endif

25  #include <esl/c_portable.h>
26  #include <esl/ep_xopen.h>
27  #include <esl/inout.h>

29  #include <stdlib.h>
30  #include <sys/types.h>
31  #include <pthread.h>

33  #include <logging/logging.h>
34  #include <csc/cscomm.h>
35  #include <eerrno/e_eb.h>


38  // Rogue Wave includes
39  #include <rw/collect.h>
40  #include <rw/rwfile.h>
41  #include <rw/vstream.h>
42  #include <rw/gqueue.h>

44  #ifdef __cplusplus
45 1 extern "C" {
46 1 #endif

48 1 #include <restore/dispatch_daemon.h>

50 1 #ifdef __cplusplus
51  }
52  #endif

54  #include <EDMSession.h>
55  #include <EDMDDHandleMgrApi.h>
56  #include <EDMReturnMessage.h>
57  #include <EDMReturnMessageApi.h>

59  declare(RWGQueue, EDMReturnMessage)

61  RWGQueue(EDMReturnMessage) g_messageQueue;

63  static pthread_mutex_t  G_returnmessagemutex =
                                                  PTHREAD_MUTEX_INITIALIZER;
```



```
65  /*************************************************************************
                                                                    ********
66  **
67  ** Routine:  LockReturnMessageMutex
68  **
69  ** Inputs:   None
70  **
71  ** Outputs:  None
72  **
73  ** Return Codes:
74  **           None
75  **
76  ** Purpose:  Lock the mutex for the return message object
77  **
78  *************************************************************************
                                                                    ********
79  */

81  static void
82  LockReturnMessageMutex()
83 1 {
84 1     static boolean_ty first = TRUE;

86 1     if (first == TRUE)
87 2     {
88 2         first = FALSE;
89 2         pthread_mutex_init(&G_returnmessagemutex, NULL);
90 1     }

92 1     pthread_mutex_lock(&G_returnmessagemutex);
93  }

95  /*************************************************************************
                                                                    ********
96  **
97  ** Routine:  UnlockReturnMessageMutex
98  **
99  ** Inputs:   None
100 **
101 ** Outputs:  None
102 **
103 ** Return Codes:
104 **           None
105 **
106 ** Purpose:  Unlock the mutex for the return message object
107 **
108 *************************************************************************
                                                                    ********
109 */

111 static void
112 UnlockReturnMessageMutex()
113 1 {
114 1     pthread_mutex_unlock(&G_returnmessagemutex);
115  }

117 /*************************************************************************
                                                                    ********
118 **
119 ** Routine:  PushResponseMessage
120 **
121 ** Inputs:   int msgtoqueue - an integer representing the message to
                                                                      queue.
122 **           DD_client_session_id sid - Source service of message.
123 **
```

```
124   ** Outputs:  int *status - a status for the push command
125   **
126   ** Return Codes:
127   **           0 for success and non-zero for failure.
128   **
129   ** Purpose:  Push a message on the Message Queue
130   **
131   ************************************************************************
                                                                        ********
132   */

134   int
135   PushResponseMessage(IN int msgtoqueue,
136                       IN DD_client_session_id sid,
137                      IN rpc_binding_handle_t *csc_h,
138                      OUT int *status)
139 1 {
140 1     EDMReturnMessage *msg, *ret;

142 1     if (status == NULL)
143 2     {
144 2         return 1;
145 1     }

147 1     // At one point we did check the csc_h (
                                                CSC handle) but we found that
148 1     // we would rather take a NULL and check it once we try to use it.
149 1     // That way we can handle cases where the private service messages
                                                                        us
150 1     // and we haven't made a connection to it yet.

152 1     if (msgtoqueue == 0)
153 2     {
154 2         *status = RETURNMESSAGE_BAD_PARAM;
155 2         return 1;
156 1     }

158 1     //
159 1     // Acquire a new message Object.
160 1     //
161 1     msg = new EDMReturnMessage();
162 1     if (msg == NULL)
163 2     {
164 2         *status = RETURNMESSAGE_NO_MEMORY;
165 2         return(1);
166 1     }

168 1     //
169 1     // Build the message
170 1     //
171 1     msg -> setSessionID(sid);
172 1     msg -> setMessage(msgtoqueue);
173 1     msg -> setTimeIssued(time(NULL));
174 1     msg -> setBindingHandle(csc_h);

176 1     LockReturnMessageMutex();

178 1     ret = g_messageQueue.append(msg);

180 1     UnlockReturnMessageMutex();

182 1     if (ret == NULL)
183 2     {
184 2         *status = RETURNMESSAGE_QUEUE_APPEND_FAILURE;
185 2         delete msg;
186 2         return 1;
```



```
187 1     }

189 1     return 0;
190   }

192   /************************************************************************
                                                                        ********

193   **
194   ** Routine:  PopResponseMessage
195   **
196   ** Inputs:
197   **
198   ** Outputs:  int *status - a status for the pop message
199   **           int *msgid   - a place to put the message to queue
200   **           DD_client_session_id *sess - a place to put the
201   **                        session ID
202   **            rpc_binding_handle_t **client_h_p - a handle to respond
                                                                        on.
203   **
204   ** Return Codes:
205   **           0 for success and non-zero for failure.
206   **
207   ** Purpose:  Pop a message off the Message Queue.
                                                  If the Queue is empty
208   **           block until a signal wakes us up. If the timeout value is
209   **           0 the thread blocks until a message is received,
                                                                  otherwise
210   **           it will block the length of time specified.
211   **
212   ************************************************************************
                                                                        ********

213   */

215   int
216   PopResponseMessage(
                OUT int *ResponseMessage, OUT DD_client_session_id *sess,
217                     OUT rpc_binding_handle_t **client_h_p,
                                                          OUT int *status)

218 1 {
219 1     EDMReturnMessage *ret;

221 1     if (status == NULL)
222 2     {
223 2         return 1;
224 1     }

226 1     *status = 0;

228 1     if (client_h_p == NULL || ResponseMessage == NULL || sess == NULL)
229 2     {
230 2         *status = RETURNMESSAGE_BAD_PARAM;
231 2         return -1;
232 1     }

234 1     LockReturnMessageMutex();

236 1     if (g_messageQueue.isEmpty())
237 2     {
238 2         *status = RETURNMESSAGE_QUEUE_IS_EMPTY;
239 2         UnlockReturnMessageMutex();
240 2          return -1;
241 1     }

243 1     ret = g_messageQueue.get();

245 1     UnlockReturnMessageMutex();
```

```
247  1      if (ret == NULL)
248  2      {
249  2          *status = RETURNMESSAGE_RECORD_GET_FAILED;
250  2          return -1;
251  1      }

253  1      ret -> getSessionID(sess);
254  1      *ResponseMessage = ret -> getMessage();
255  1      *client_h_p = ret -> getBindingHandle();

257  1      delete ret;

259  1      return 0;
260    }
```

```
/*****************************************************************************
                                                                          ******
**
** File Name:   EDMDispProtocolSvc.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
**      This module contains the callback functions for use with
**       the dispatch daemon protocol.
**
**Table of Contents:
** ------------------
**
**
** Compile-Time Options:
**              This section must list any compile time definitions
**              which will affect this header.
**
** Copyright (c) 1996 by EMC Corp.
**
*****************************************************************************
                                                                          *****/

#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMDispProtocolSvc.c,v $ "
                            "$Revision: 1.0 $ "
                            "$Date: 1999/03/06 09:00:00 $" ;
#endif

#include <esl/c_portable.h>
#include <esl/inout.h>
#include <util/esl_string.h>

#include <logging/logging.h>
#include <csc/cscomm.h>
#include <eerrno/e_eb.h>

#include <sys/time.h>
#include <pthread.h>
#include <EDMccw.h>
#include <EDMDD_ddp.h>
#include <EDMutils.h>

#ifdef __cplusplus
extern "C" {
#endif

#include <restore/dispatch_daemon.h>
#include <restore/dispatch_protocol.h>
#include <restore/dispatch_protocol_service.h>
#include <restore/dispatch_protocol_client.h>
#include <restore/csc_Dispatch_Protocol_Service.h>

#include <EDMDispatchSession.h>
#include <EDMTimedMessageApi.h>
#include <EDMReturnMessageApi.h>
#include <EDMDDHandleMgrApi.h>

#include <logging/logging.h>
#include <EDMDispatchLog.h>

#ifdef __cplusplus
}
#endif

/*****************************************************************************
```



```
                                                                          ******
**
**  Structures
**
*****************************************************************************
                                                                          ******

*/

/*****************************************************************************
                                                                          ******

**
** Routine:  dp_connect_indicate_1()
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:
**
** Intended caller:  Internal Only.
*****************************************************************************
                                                                          ******

*/
int *
dp_connect_indicate_1_svc(
                    DP_connect_indicate_msg *msg,struct svc_req *req)
{
 int rc;
 int status;
 rpc_binding_handle_t *client_handle_p = NULL;

  /* Update last time we herd from the service */
  rc = UpdateSessionLastReceived( &msg->sid );
  if (0 != rc)
  {
     EDMDispatch_logent(
        __FILE__,__LINE__, LOG_ERR, DDP_UPDATE_SESSION_RCV_FAILURE, 0,
                        "UpdateSessionLastReceived failed.");
  }

  /* Get the csc_binding_handle associated with this sid */
  rc = GetCSCHandle(&msg->sid,
                    &client_handle_p,
                    &status );
  if (0 != rc)
  {
     EDMDispatch_logent(
      __FILE__,__LINE__, LOG_ERR, DDP_GET_CSC_HANDLE_FAILURE, status,
                        "GetCSCHandle failed.");
  }

  /* Push message to send onto the queue */
  rc = PushResponseMessage((int) dp_connect_confirm,
                           msg->sid,
                           client_handle_p,
                           &status);
  if (0 != rc)
   {
      EDMDispatch_logent(
         __FILE__,__LINE__, LOG_ERR, DDP_PUT_RESPONSE_FAILURE, status,
                        "PushResponseMessage failed.");
   }
```

```
123 1    if (IsDebugOn())
124 2    {
125 2         (void) EDMDispatch_logent(
                        __FILE__,__LINE__, LOG_ERR, DDP_SENDING_MESSAGE, 0,
126 2                        "Pushing response message to restore service.
                                                                        ");

127 1    }

129 1   return((int*)0);
130     }
```



```
132     /*************************************************************************
                                                                        ********

133     **
134     ** Routine:  dp_abort_response_1()
135     **
136     ** Inputs:   None
137     **
138     ** Outputs:  None
139     **
140     ** Return Codes:
141     **           None
142     **
143     ** Purpose:
144     **
145     ** Intended caller:  Internal Only.
146     *************************************************************************
                                                                        ********

147     */
148     int *
149     dp_abort_response_1_svc(
                              DP_abort_response_msg *msg,struct svc_req *req)

150 1   {
151 1    int rc;
152 1    int status;


155 1     /* Update last time we herd from the service */
156 1     rc = UpdateSessionLastReceived( &msg->sid );
157 1     if (0 != rc)
158 2      {
159 2          EDMDispatch_logent(
                 __FILE__,__LINE__, LOG_ERR, DDP_UPDATE_SESSION_RCV_FAILURE, 0,
160 2                              "UpdateSessionLastReceived failed.");
161 1      }

163 1     /* Remove the timed message to indicate that we got the response */
164 1     rc = deleteTimedMessage(&msg->sid,
165 1                             dp_abort_request,
166 1                             &status);
167 1     if (0 != rc)
168 2      {
169 2          EDMDispatch_logent(
               __FILE__,__LINE__, LOG_ERR, DDP_DELETE_TIMED_MSG_FAILURE, status,
170 2                              "deleteTimedMessage failed.");
171 1      }

173 1    return( (int*)0 );
174     }
```

```
176  /*****************************************************************************
                                                                          ********
177  **
178  ** Routine:  dp_close_response_1()
179  **
180  ** Inputs:   None
181  **
182  ** Outputs:  None
183  **
184  ** Return Codes:
185  **           None
186  **
187  ** Purpose:
188  **
189  ** Intended caller:  Internal Only.
190  *****************************************************************************
                                                                          ********
191  */

193  int *
194  dp_close_response_1_svc(
                         DP_close_response_msg *msg,struct svc_req *req)
195 1 {
196 1  int rc;
197 1  int status;

199 1   /* Update last time we herd from the service */
200 1   rc = UpdateSessionLastReceived( &msg->sid );
201 1   if (0 != rc)
202 2    {
203 2       EDMDispatch_logent(
             __FILE__,__LINE__, LOG_ERR, DDP_UPDATE_SESSION_RCV_FAILURE, 0,
204 2                          "UpdateSessionLastReceived failed.");
205 1    }

207 1   /* Remove the timed message to indicate that we got the response */
208 1   rc = deleteTimedMessage(&msg->sid,
209 1                           dp_close_request,
210 1                           &status);
211 1   if (0 != rc)
212 2    {
213 2       EDMDispatch_logent(
           __FILE__,__LINE__, LOG_ERR, DDP_DELETE_TIMED_MSG_FAILURE, status,
214 2                          "deleteTimedMessage failed.");
215 1    }

217 1  return( (int*)0 );
218  }
```



```
219  /*****************************************************************************
                                                                          ********
220  **
221  ** Routine:  dp_ping_response_1()
222  **
223  ** Inputs:   None
224  **
225  ** Outputs:  None
226  **
227  ** Return Codes:
228  **           None
229  **
230  ** Purpose:
231  **
232  ** Intended caller:  Internal Only.
233  *****************************************************************************
                                                                          ********
234  */

236  int *
237  dp_ping_response_1_svc( DP_ping_response_msg *msg,struct svc_req *req)
238 1 {
239 1  int rc;
240 1  int status;

242 1   /* Update last time we herd from the service */
243 1   rc = UpdateSessionLastReceived( &msg->sid );
244 1   if (0 != rc)
245 2    {
246 2       EDMDispatch_logent(
             __FILE__,__LINE__, LOG_ERR, DDP_UPDATE_SESSION_RCV_FAILURE, 0,
247 2                          "UpdateSessionLastReceived failed.");
248 1    }

250 1   /* Remove the timed message to indicate that we got the response */
251 1   rc = deleteTimedMessage(&msg->sid,
252 1                           dp_ping_request,
253 1                           &status);
254 1   if (0 != rc)
255 2    {
256 2       EDMDispatch_logent(
           __FILE__,__LINE__, LOG_ERR, DDP_DELETE_TIMED_MSG_FAILURE, status,
257 2                          "deleteTimedMessage failed.");
258 1    }

260 1  return( (int*)0 );
261  }
```

```
263  /*****************************************************************************
                                                                          ********
264  **
265  ** Routine:  dp_event_indicate_1()
266  **
267  ** Inputs:   None
268  **
269  ** Outputs:  None
270  **
271  ** Return Codes:
272  **           None
273  **
274  ** Purpose:
275  **
276  ** Intended caller:  Internal Only.
277  *****************************************************************************
                                                                          ********
278  */

280  int *
281  dp_event_indicate_1_svc(
                        DP_event_indicate_msg *msg,struct svc_req *req)
282  1 {
283  1  int rc;
284  1  int status;
285  1  rpc_binding_handle_t *client_handle_p;

287  1    /* Update last time we herd from the service */
288  1    rc = UpdateSessionLastReceived( &msg->sid );
289  1    if (0 != rc)
290  2     {
291  2        EDMDispatch_logent(
               __FILE__,__LINE__, LOG_ERR, DDP_UPDATE_SESSION_RCV_FAILURE, 0,
292  2                           "UpdateSessionLastReceived failed.");
293  1     }

295  1  return( (int*)0 );
296  }
```



```
297  /*****************************************************************************
                                                                          ********
298  **
299  ** Routine:  dp_progress_indicate_1()
300  **
301  ** Inputs:   None
302  **
303  ** Outputs:  None
304  **
305  ** Return Codes:
306  **           None
307  **
308  ** Purpose:
309  **
310  ** Intended caller:  Internal Only.
311  *****************************************************************************
                                                                          ********
312  */

314  int *
315  dp_progress_indicate_1_svc(
                        DP_progress_indicate_msg *msg,struct svc_req *req)
316  1 {
317  1  int rc;
318  1  int status;

320  1  /* Update last time we herd from the service */
321  1  rc = UpdateSessionLastReceived( &msg->sid );
322  1  if (0 != rc)
323  2    {
324  2       EDMDispatch_logent(
             __FILE__,__LINE__, LOG_ERR, DDP_UPDATE_SESSION_RCV_FAILURE, 0,
325  2                           "UpdateSessionLastReceived failed.");
326  1    }

328  1  return( (int*)0 );
329  }
```

```
331  /*****************************************************************
                                                                  ********
332  **
333  ** Routine:  dp_final_stats_indicate()
334  **
335  ** Inputs:   None
336  **
337  ** Outputs:  None
338  **
339  ** Return Codes:
340  **           None
341  **
342  ** Purpose:
343  **
344  ** Intended caller:  Internal Only.
345  *****************************************************************
                                                                  ********
346  */

348  int *
349  dp_final_stats_indicate_1_svc(DP_final_stats_indicate_msg *msg,
350                                struct svc_req *req)
351  1 {
352  1  int rc;
353  1  int status;
354  1  rpc_binding_handle_t *client_handle_p = NULL;

356  1  /* Update last time we herd from the service */
357  1  rc = UpdateSessionLastReceived( &msg->sid );
358  1  if (0 != rc)
359  2  {
360  2       EDMDispatch_logent(
             __FILE__,__LINE__, LOG_ERR, DDP_UPDATE_SESSION_RCV_FAILURE, 0,
361  2                          "UpdateSessionLastReceived failed.");
362  1  }
363  1  /* Get the csc_binding_handle associated with this sid */
364  1  rc = GetCSCHandle(&msg->sid,
365  1                    &client_handle_p,
366  1                    &status );
367  1  if (0 != rc)
368  2  {
369  2       EDMDispatch_logent(
             __FILE__,__LINE__, LOG_ERR, DDP_GET_CSC_HANDLE_FAILURE, status,
370  2                          "GetCSCHandle failed.");
371  1  }

373  1  /* Push message to send onto the queue */
374  1  rc = PushResponseMessage(dp_final_stats_confirm,
375  1                           msg->sid,
376  1                           client_handle_p,
377  1                           &status);
378  1  if (0 != rc)
379  2  {
380  2       EDMDispatch_logent(
             __FILE__,__LINE__, LOG_ERR, DDP_PUT_RESPONSE_FAILURE, status,
381  2                          "PushResponseMessage failed.");
382  1  }

384  1  return( (int*)0 );
385  }
```